Scipy Cookbook

wizardforcel

Published with GitBook



Table of Contents

SciPy Cookbook

From: SciPy Cookbook

SciPy Cookbook

This is the "SciPy Cookbook" — a collection of various user-contributed recipes, which once lived under `wiki.scipy.org`. Note that some are fairly old (2005–2009), and may not be as relevant today. If you want to contribute additions/corrections, see the the SciPy-CookBook repository.

Compiling Extensions

Compiling Extension Modules on Windows using mingw

Graphics

Line Integral Convolution Mayavi Vtk volume rendering

Input Output

Data Acquisition with NIDAQmx Data acquisition with PyUL Fortran I/O Formats Input and output LAS reader Reading SPE file from CCD camera Reading mat files hdf5 in Matlab

Matplotlib / 3D Plotting

Matplotlib VTK integration Matplotlib: mplot3d

Matplotlib / Embedding Plots in Apps

Embedding In Wx Matplotlib: pyside Matplotlib: scrolling plot

Matplotlib / Misc

Load image Matplotlib: adjusting image size Matplotlib: compiling matplotlib on solaris10 Matplotlib: deleting an existing data series Matplotlib: django Matplotlib: interactive plotting Matplotlib: matplotlib and zope Matplotlib: multiple subplots with one axis label Matplotlib: qt with ipython and designer Matplotlib: using matplotlib in a CGI script

Matplotlib / Pseudo Color Plots

Matplotlib / Simple Plotting

Matplotlib / Typesetting

Mayavi

Mayavi / TVTK

Numpy & Scipy / Advanced topics

Numpy & Scipy / Interpolation

Numpy & Scipy / Linear Algebra

Numpy & Scipy / Matplotlib

Histograms

Numpy & Scipy / Optimization and fitting techniques

Fitting data Large-scale bundle adjustment in scipy Least squares circle Linear regression OLS Optimization and fit demo Optimization demo RANSAC Robust nonlinear regression in scipy Solving a discrete boundary-value problem in scipy

Numpy & Scipy / Ordinary differential equations

Coupled spring-mass system Korteweg de Vries equation Matplotlib: lotka volterra tutorial Modeling a Zombie Apocalypse Theoretical ecology: Hastings and Powell

Numpy & Scipy / Other examples

Applying a FIR filter Brownian Motion Butterworth Bandpass Communication theory Correlated Random Samples Easy multithreading Eye Diagram FIR filter Filtfilt Finding the Convex Hull of a 2-D Dataset Finding the minimum point in the convex hull of a finite set of points Frequency swept signals KDTree example Kalman filtering Linear classification Particle filter Rebinning Savitzky Golay Filtering Smoothing of a 1D signal Solving large Markov Chains Watershed

Numpy & Scipy / Root finding

Function intersections Spherical Bessel Zeros

Numpy & Scipy / Tips and tricks

Addressing Array Columns by Name Building arrays Convolution-like operations Indexing numpy arrays MetaArray Multidot Object arrays using record arrays Stride tricks for the Game of Life accumarray like function

Other examples

C Extensions for Using NumPy Arrays Embedding in Traits GUI Matplotlib: drag’n’drop text example Matplotlib: treemap Mayavi: Install python stuff from source Mayavi: examples Reading custom text files with Pyparsing Scripting Mayavi 2: basic modules Scripting Mayavi 2: filters Scripting Mayavi 2: main modules

Performance

A beginners guide to using Python for performance computing Parallel Programming with numpy and scipy

Scientific GUIs

wxPython dialogs

Scientific Scripts

FDTD Algorithm Applied to the Schrödinger Equation

Using NumPy With Other Languages (Advanced)

C extensions Ctypes F2py Inline Weave With Basic Array Conversion (no Blitz) Pyrex And NumPy SWIG Numpy examples SWIG and Numpy SWIG memory deallocation f2py and numpy

Outdated

A numerical agnostic pyrex class Array, struct, and Pyrex Data Frames Python Imaging Library Recipes for timeseries The FortranFile class dbase xplt

Compiling Extensions

Compiling Extension Modules on Windows using mingw

Pre-reqs

1. Python 2.5 (with earlier versions of Python, it is much more complicated to compile extensions with mingw. Use visual Studio 2003 if you cannot upgrade to Python 2.5)
2. Local admin access on your machine

Compiler setup

1. Go to www.mingw.org and go to the “downloads” section of the web-site. Look for the link to the sourceforge page and go there. Grab the latest version of the “Automated MinGW Installer” and run the installer.

2. When asked “which MinGW package” you would like to install, select “Current” or “Candidate” depending on whether you want to be on the bleeding edge or not.

3. When you get to the section where you need to select the various components you want to install, include the “base tools” and the “g++ compiler”, and possibly the g77 compiler too if you are going to be compiling some extensions that use Fortran.

4. Install it to “C:” (the default location)

5. Add “C:” to your PATH environment variable if the installer did not do this for you. If you are running Vista, you will need to add “C:32.4.5” to your PATH as well

6. In notepad (or your favourite text editor) create a new text file and enter the following in the file:

   [build] compiler = mingw32

7. Save the file as “C:25.cfg”. This will tell python to use the MinGW compiler when compiling extensions

8. Close down any command prompts you have open (they need to be reopened to see the new environment variable values)

Compiling an extension (through distutils)

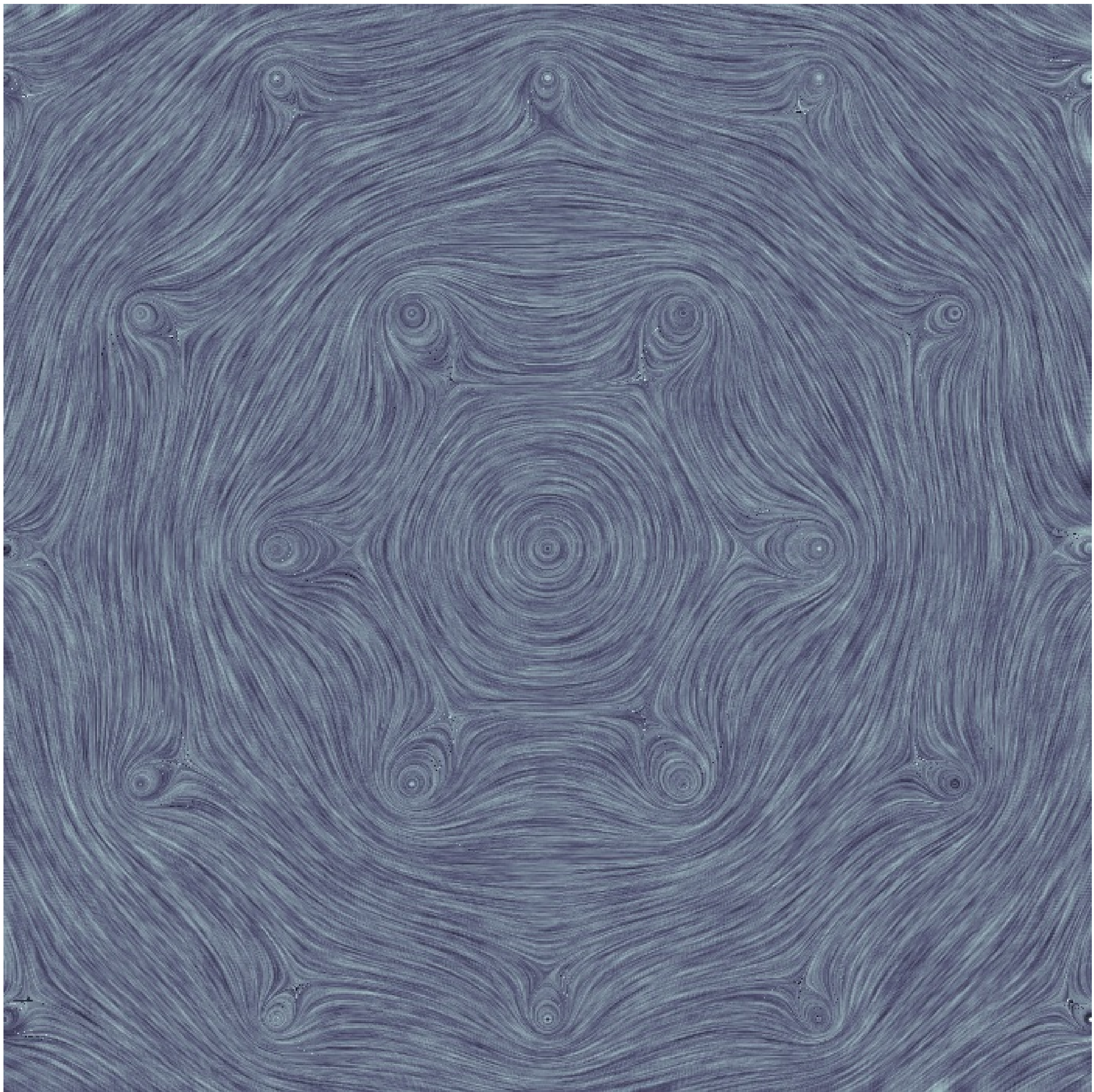
Note: this will only work for modules which have a setup.py script that uses distutils

1. So lets say you have downloaded code for some module to the folder “xyz”. Open a command prompt and cd to this directory (eg. type “c:” then press enter to change to the C drive, then type “cd c:” to change to directory xyz)
2. To create a binary installer, type “python setup.py bdist_wininst”
3. If everything worked fine, then there should now be a subfolder called “dist” which contains an exe file that you can run to install the module.

Graphics

Line Integral Convolution

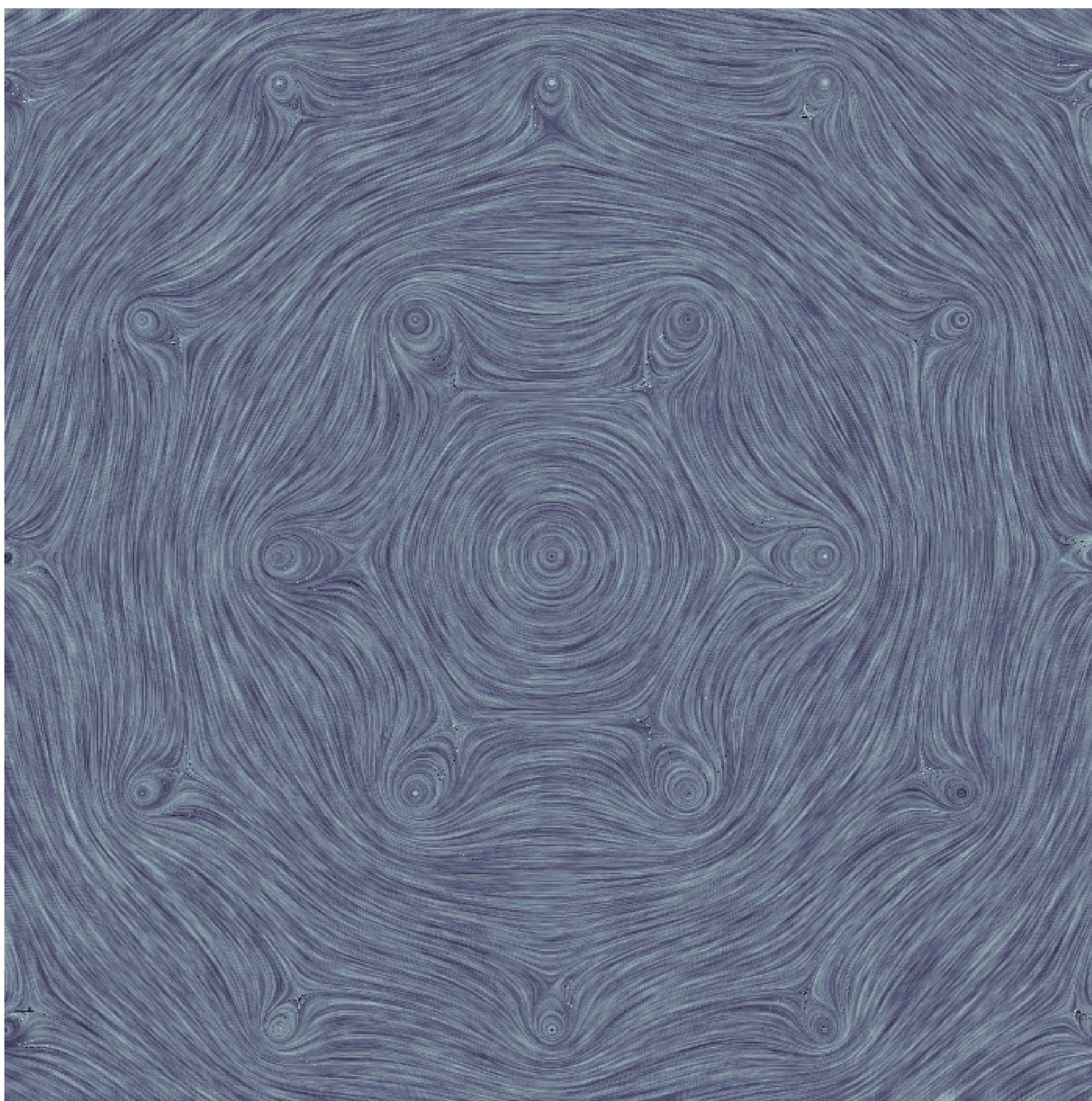
Line integral convolution is a technique, or family of techniques, for representing two-dimensional vector fields. The idea is to produce a texture which is highly correlated in the direction of the vector field but not correlated across the vector field. This is done by generating a noise texture then, for each pixel of the image, “flowing” forward and back along the vector field. The points along this path are looked up in the noise texture and averaged to give the LIC texture at the starting point. The basic technique ignores both the magnitude of the vector field and its sign. With a minor modification the same technique can be used to produce an animation of “flow” along the vector field.



Attached to this page is cython code to implement a simple line integral convolution operator, plus some demonstration python code. The demo code can either make more or less the image above - a simple array of vortices; note how an overall rotation appears in the sum of individual vortex vector fields, just as a superfluid's "bulk rotation" is actually a vortex array - or it can make a video of the same vector field. The video is a little awkward to work with, since all the standard video compression techniques butcher it horribly, but it does work well.

Attachments

- `flow-image.png`
- `lic.py`
- `lic_demo.py`
- `lic_internal.pyx`
- `setup.py`



Mayavi

Mayavi2 is a interactive program allowing elaborate 3D plots of scientific data in Python with scipy. It is the successor of MayaVi for 3D visualization.

||<#80FF80> The information on this page is a bit old and outdated. You are invited to refer to the Mayavi2 user guide as a reference. For a quick introduction to scripting, see mlab. In recent versions of Mayavi2, the user guide can be accessed in the help menu, but the user guide for the latest version of Mayavi can also be found on line. ||

If you need more help, you are invited to ask questions on the Enthought-dev mailing list.

```
#!figure
#class right
## Snazzy graphics here...
# ![](files/attachments/MayaVi/mayavi2.png

!Mayavi2 relies on [http://www.vtk.org VTK], and especially a pytho

A mayavi2 session.
```

MayaVi2 topics

!MayaVi2 can be used as an interactive program or not, as it will be presented here.

```
* For installation of !MayaVi2 in the enthought tool suite see the
documentation
```

<http://enthought.github.com/mayavi/mayavi/installation.html>`__

```
* Using !MayaVi2:
```

```
There are (at least) two ways to use !MayaVi2:
```

```
* [:Cookbook/MayaVi/RunningMayavi2: Running MayaVi2] on the command
```

```
* [:Cookbook/MayaVi/ScriptingMayavi2: Scripting MayaVi2] in Python.
```

```
* [:Cookbook/MayaVi/Examples: Scripting Examples] (all provided in
```

```
* Using Contour Module (contour.py)
```

```
* Using Glyph Module (glyph.py)
```

```
* Using Mayavi2 without GUI (nongui.py)
```

```
* A 3D array as numerical source (numeric_source.py)
```

```
* Using Streamline Module (streamline.py)
```

```
* Using !ImagePlaneWidget Module (test.py)
```

```
* Plotting a surface from a matrix (surf_regular_mlab.py). See also
```

```
* [:Cookbook/MayaVi/Tips: Tips]: General tips for !MayaVi2 and arour
```

Vtk volume rendering

As I had some problems in figuring out how to use VTK to render data contained in a three dimensional numpy array, I have decided to share my code. This code is based on the otherwise excellent documentation for VTK and the now outdated vtkImageImportFromArray-class created by David Gobbi found at http://public.kitware.com/cgi-bin/cvsweb.cgi/vtk/python/?cvsroot=vtk

The example is very simple, for more advanced functionality: read the documentation.

```
import vtk
from numpy import *

# We begin by creating the data we want to render.
# For this tutorial, we create a 3D-image containing three overlap
# This data can of course easily be replaced by data from a medica
# The only limit is that the data must be reduced to unsigned 8 bi
data_matrix = zeros([75, 75, 75], dtype=uint8)
data_matrix[0:35, 0:35, 0:35] = 50
data_matrix[25:55, 25:55, 25:55] = 100
data_matrix[45:74, 45:74, 45:74] = 150

# For VTK to be able to use the data, it must be stored as a VTK-in
# imports raw data and stores it.
dataImporter = vtk.vtkImageImport()
# The preaviusly created array is converted to a string of chars an
data_string = data_matrix.tostring()
dataImporter.CopyImportVoidPointer(data_string, len(data_string))
# The type of the newly imported data is set to unsigned char (uin
dataImporter.SetDataScalarTypeToUnsignedChar()
# Because the data that is imported only contains an intensity val
# must be told this is the case.
dataImporter.SetNumberOfScalarComponents(1)
# The following two functions describe how the data is stored and
# simple case, all axes are of length 75 and begins with the first
# I have to admit however, that I honestly dont know the differenc
# VTK complains if not both are used.
dataImporter.SetDataExtent(0, 74, 0, 74, 0, 74)
dataImporter.SetWholeExtent(0, 74, 0, 74, 0, 74)

# The following class is used to store transparencyv-values for la
# completly opaque whereas the three different cubes are given dif
alphaChannelFunc = vtk.vtkPiecewiseFunction()
alphaChannelFunc.AddPoint(0, 0.0)
alphaChannelFunc.AddPoint(50, 0.05)
alphaChannelFunc.AddPoint(100, 0.1)
alphaChannelFunc.AddPoint(150, 0.2)
```

```
# This class stores color data and can create color tables from a f
# to be of the colors red green and blue.
colorFunc = vtk.vtkColorTransferFunction()
colorFunc.AddRGBPoint(50, 1.0, 0.0, 0.0)
colorFunc.AddRGBPoint(100, 0.0, 1.0, 0.0)
colorFunc.AddRGBPoint(150, 0.0, 0.0, 1.0)

# The preavius two classes stored properties. Because we want to ap
# we have to store them in a class that stores volume prpoperties.
volumeProperty = vtk.vtkVolumeProperty()
volumeProperty.SetColor(colorFunc)
volumeProperty.SetScalarOpacity(alphaChannelFunc)

# This class describes how the volume is rendered (through ray trac
compositeFunction = vtk.vtkVolumeRayCastCompositeFunction()
# We can finally create our volume. We also have to specify the dat
volumeMapper = vtk.vtkVolumeRayCastMapper()
volumeMapper.SetVolumeRayCastFunction(compositeFunction)
volumeMapper.SetInputConnection(dataImporter.GetOutputPort())

# The class vtkVolume is used to pair the preaviusly declared volum
volume = vtk.vtkVolume()
volume.SetMapper(volumeMapper)
volume.SetProperty(volumeProperty)

# With almost everything else ready, its time to initialize the ren
renderer = vtk.vtkRenderer()
renderWin = vtk.vtkRenderWindow()
renderWin.AddRenderer(renderer)
renderInteractor = vtk.vtkRenderWindowInteractor()
renderInteractor.SetRenderWindow(renderWin)

# We add the volume to the renderer ...
renderer.AddVolume(volume)
# ... set background color to white ...
renderer.SetBackground(1, 1, 1)
# ... and set window size.
renderWin.SetSize(400, 400)

# A simple function to be called when the user decides to quit the
def exitCheck(obj, event):
    if obj.GetEventPending() != 0:
        obj.SetAbortRender(1)

# Tell the application to use the function as an exit check.
renderWin.AddObserver("AbortCheckEvent", exitCheck)

renderInteractor.Initialize()
# Because nothing will be rendered without any input, we order the
renderWin.Render()
renderInteractor.Start()
```

To exit the application, simply press *q*.

In my opinion, the volume renderer creates extremely ugly images if not the following option is used:

```
volumeProperty.ShadeOn()
```

Input Output

Data Acquisition with NIDAQmx

These are quick examples of using ctypes and numpy to do data acquisition and playback using National Instrument's NI-DAQmx library. This library allows access to their wide range of data acquisition devices. By using ctypes, we bypass the need for a C compiler. The code below assumes a Windows platform. NI-DAQmx is also available for Linux, but the code below would require a few minor changes, namely loading the shared library and setting the function signatures.

See also `Data acquisition with PyUniversalLibrary`.

See also projects that wrap NI-DAQmx library with Linux support: pylibnidaqmx, pydaqmx, daqmxbase-swig.

OK, enough talk, let's see the code!

Analog Acquisition

```
#!python numbers=disable
#Acq_IncClk.py
# This is a near-verbatim translation of the example program
# C:\Program Files\National Instruments\NI-DAQ\Examples\DAQmx ANSI
import ctypes
import numpy
nidaq = ctypes.windll.nicaiu # load the DLL
##############################
# Setup some typedefs and constants
# to correspond with values in
# C:\Program Files\National Instruments\NI-DAQ\DAQmx ANSI C Dev\inc
# the typedefs
int32 = ctypes.c_long
uInt32 = ctypes.c_ulong
uInt64 = ctypes.c_ulonglong
float64 = ctypes.c_double
TaskHandle = uInt32
# the constants
DAQmx_Val_Cfg_Default = int32(-1)
DAQmx_Val_Volts = 10348
DAQmx_Val_Rising = 10280
DAQmx_Val_FiniteSamps = 10178
DAQmx_Val_GroupByChannel = 0
##############################
def CHK(err):
    """a simple error checking routine"""
    if err < 0:
        buf_size = 100
        buf = ctypes.create_string_buffer('\000' * buf_size)
```

```
        nidaq.DAQmxGetErrorString(err,ctypes.byref(buf),buf_size)
        raise RuntimeError('nidaq call failed with error %d: %s'%(e
# initialize variables
taskHandle = TaskHandle(0)
max_num_samples = 1000
data = numpy.zeros((max_num_samples,),dtype=numpy.float64)
# now, on with the program
CHK(nidaq.DAQmxCreateTask("",ctypes.byref(taskHandle)))
CHK(nidaq.DAQmxCreateAIVoltageChan(taskHandle,"Dev1/ai0","",
                                   DAQmx_Val_Cfg_Default,
                                   float64(-10.0),float64(10.0),
                                   DAQmx_Val_Volts,None))
CHK(nidaq.DAQmxCfgSampClkTiming(taskHandle,"",float64(10000.0),
                                DAQmx_Val_Rising,DAQmx_Val_FiniteSa
                                uInt64(max_num_samples)));
CHK(nidaq.DAQmxStartTask(taskHandle))
read = int32()
CHK(nidaq.DAQmxReadAnalogF64(taskHandle,max_num_samples,float64(10.
                             DAQmx_Val_GroupByChannel,data.ctypes.d
                             max_num_samples,ctypes.byref(read),Nor
print "Acquired %d points"%(read.value)
if taskHandle.value != 0:
    nidaq.DAQmxStopTask(taskHandle)
    nidaq.DAQmxClearTask(taskHandle)
print "End of program, press Enter key to quit"
raw_input()
```

Analog Generation

```
#!python numbers=disable
"""
This is an interpretation of the example program
C:\Program Files\National Instruments\NI-DAQ\Examples\DAQmx ANSI C\
This routine will play an arbitrary-length waveform file.
This module depends on:
numpy
Adapted by Martin Bures [ mbures { @ } zoll { . } com ]
"""
# import system libraries
import ctypes
import numpy
import threading
# load any DLLs
nidaq = ctypes.windll.nicaiu # load the DLL
##############################
# Setup some typedefs and constants
# to correspond with values in
# C:\Program Files\National Instruments\NI-DAQ\DAQmx ANSI C Dev\inc
# the typedefs
```

```
int32 = ctypes.c_long
uInt32 = ctypes.c_ulong
uInt64 = ctypes.c_ulonglong
float64 = ctypes.c_double
TaskHandle = uInt32
# the constants
DAQmx_Val_Cfg_Default = int32(-1)
DAQmx_Val_Volts = 10348
DAQmx_Val_Rising = 10280
DAQmx_Val_FiniteSamps = 10178
DAQmx_Val_ContSamps = 10123
DAQmx_Val_GroupByChannel = 0
##############################
class WaveformThread( threading.Thread ):
    """
 This class performs the necessary initialization of the DAQ hardwa
 spawns a thread to handle playback of the signal.
 It takes as input arguments the waveform to play and the sample ra
 to play it.
 This will play an arbitrary-length waveform file.
 """
    def __init__( self, waveform, sampleRate ):
        self.running = True
        self.sampleRate = sampleRate
        self.periodLength = len( waveform )
        self.taskHandle = TaskHandle( 0 )
        self.data = numpy.zeros( ( self.periodLength, ), dtype=nump
        # convert waveform to a numpy array
        for i in range( self.periodLength ):
            self.data[ i ] = waveform[ i ]
        # setup the DAQ hardware
        self.CHK(nidaq.DAQmxCreateTask("",
                          ctypes.byref( self.taskHandle )))
        self.CHK(nidaq.DAQmxCreateAOVoltageChan( self.taskHandle,
                                   "Dev1/ao0",
                                   "",
                                   float64(-10.0),
                                   float64(10.0),
                                   DAQmx_Val_Volts,
                                   None))
        self.CHK(nidaq.DAQmxCfgSampClkTiming( self.taskHandle,
                                "",
                                float64(self.sampleRate),
                                DAQmx_Val_Rising,
                                DAQmx_Val_FiniteSamps,
                                uInt64(self.periodLength)));
        self.CHK(nidaq.DAQmxWriteAnalogF64( self.taskHandle,
                              int32(self.periodLength),
                              0,
                              float64(-1),
                              DAQmx_Val_GroupByChannel,
                              self.data.ctypes.data,
                              None,
```

```
                                None))
        threading.Thread.__init__( self )
    def CHK( self, err ):
        """a simple error checking routine"""
        if err < 0:
            buf_size = 100
            buf = ctypes.create_string_buffer('\000' * buf_size)
            nidaq.DAQmxGetErrorString(err,ctypes.byref(buf),buf_siz
            raise RuntimeError('nidaq call failed with error %d: %s
        if err > 0:
            buf_size = 100
            buf = ctypes.create_string_buffer('\000' * buf_size)
            nidaq.DAQmxGetErrorString(err,ctypes.byref(buf),buf_siz
            raise RuntimeError('nidaq generated warning %d: %s'%(er
    def run( self ):
        counter = 0
        self.CHK(nidaq.DAQmxStartTask( self.taskHandle ))
    def stop( self ):
        self.running = False
        nidaq.DAQmxStopTask( self.taskHandle )
        nidaq.DAQmxClearTask( self.taskHandle )
if __name__ == '__main__':
    import time
    # generate a time signal 5 seconds long with 250Hz sample rate
    t = numpy.arange( 0, 5, 1.0/250.0 )
    # generate sine wave
    x = sin( t )
    mythread = WaveformThread( x, 250 )
    # start playing waveform
    mythread.start()
    # wait 5 seconds then stop
    time.sleep( 5 )
    mythread.stop()
```

Data acquisition with PyUL

Introduction

This pages illustrates the use of the inexpensive (about $150) PMD USB-1208FS data acquisition device from Measurement Computing. It makes use of PyUniversalLibrary, an open-source wrapper of Measurement Computing's Universal Library.

See also *Data acquisition with Ni-DAQmx*.

The following examples were made with PyUL Release 20050624. The pre-compiled win32 binaries of this version are compatible with the Enthought Edition of Python 2.4 (Release 1.0.0, 2006-08-02 12:20), which is what was used to run these examples.

Example 1 - Simple Analog input

The first example illustrates the use of the unbuffered analog input:

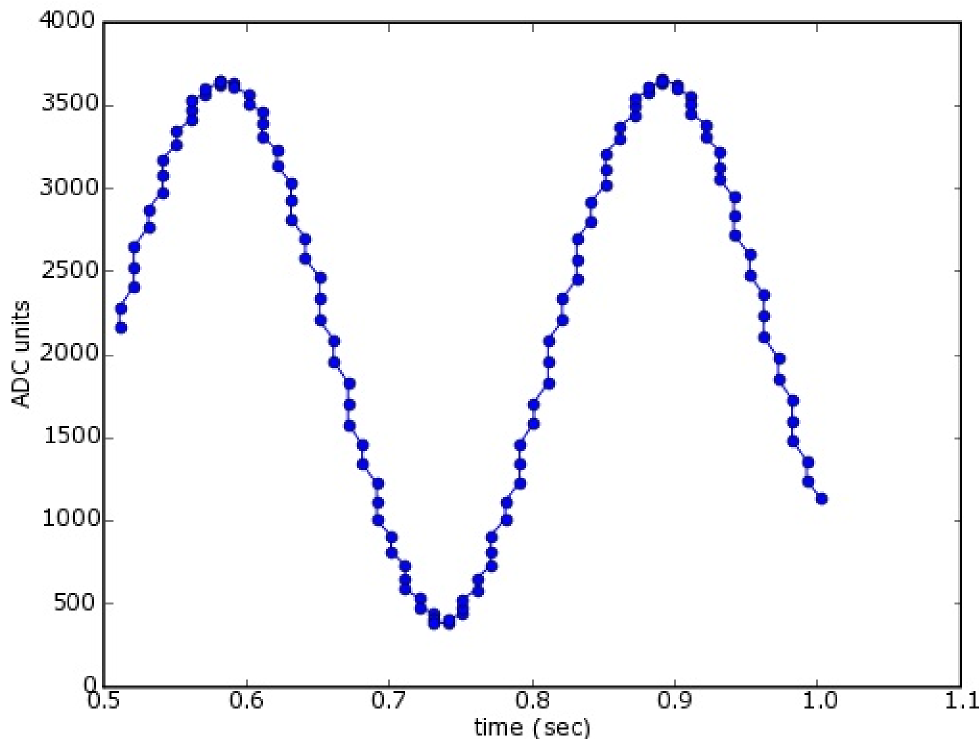
```
# example1.py
import UniversalLibrary as UL
import time

BoardNum = 0
Gain = UL.BIP5VOLTS
Chan = 0

tstart = time.time()
data = []
times = []
while 1:
    DataValue = UL.cbAIn(BoardNum, Chan, Gain)
    data.append( DataValue )
    times.append( time.time()-tstart )
    if times[-1] > 1.0:
        break

import pylab
pylab.plot(times,data,'o-')
pylab.xlabel('time (sec)')
pylab.ylabel('ADC units')
pylab.show()
```

When I ran this, I had a function generator generating a sine wave connected to pins 1 and 2 of my device. This should produce a figure like the following:



Example 2 - Getting Volts rather than arbitrary units

The values recorded in example 1 are “ADC units”, the values recorded directly by the Analog-to-Digital hardware. In fact, this device has a 12-bit A to D converter, but the values are stored as 16-bit signed integers. To convert these values to Volts, we use Measurement Computing’s function. Here we do that for each piece of data and plot the results.

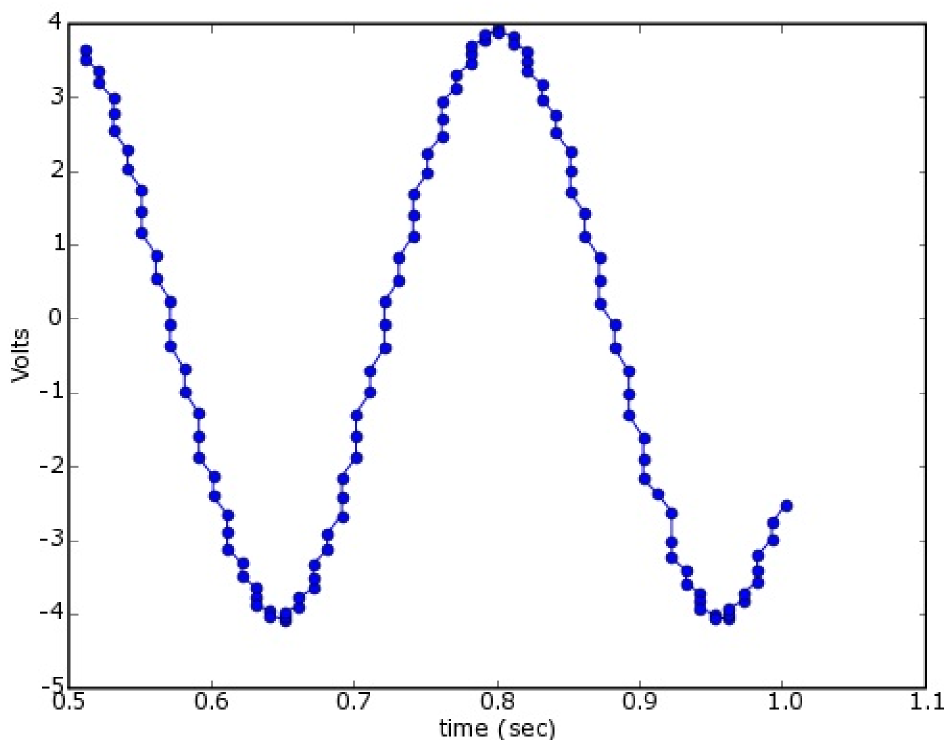
```
#example2.py
import UniversalLibrary as UL
import time

BoardNum = 0
Gain = UL.BIP5VOLTS
Chan = 0

tstart = time.time()
data = []
times = []
while 1:
    DataValue = UL.cbAIn(BoardNum, Chan, Gain)
    EngUnits = UL.cbToEngUnits(BoardNum, Gain, DataValue)
    data.append( EngUnits )
    times.append( time.time()-tstart )
    if times[-1] > 1.0:
        break

import pylab
pylab.plot(times,data,'o-')
pylab.xlabel('time (sec)')
pylab.ylabel('Volts')
#pylab.savefig('example2.png',dpi=72)
pylab.show()
```

Now the output values are in volts:



Example 3 - Buffered input

As you have no doubt noticed, the plots above aren't very "pure" sine waves. This is undoubtedly due to the way we're sampling the data. Rather than relying on a steady clock to do our acquisition, we're simply polling the device as fast as it (and the operating system) will let us go. There's a better way - we can use the clock on board the Measurement Computing device to acquire a buffer of data at evenly spaced samples.

```
#example3.py
import UniversalLibrary as UL
import Numeric
import pylab

BoardNum = 0
Gain = UL.BIP5VOLTS

LowChan = 0
HighChan = 0

Count = 2000
Rate = 3125

Options = UL.CONVERTDATA
ADData = Numeric.zeros((Count,), Numeric.Int16)

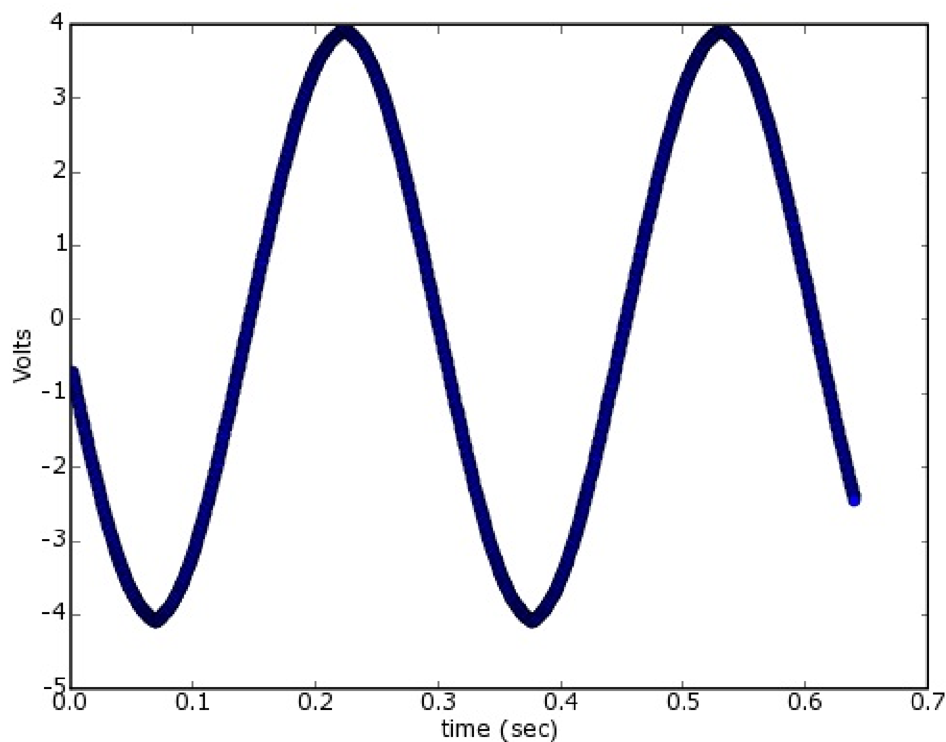
ActualRate = UL.cbAInScan(BoardNum, LowChan, HighChan, Count,
                     Rate, Gain, ADData, Options)

# convert to Volts
data_in_volts = [ UL.cbToEngUnits(BoardNum, Gain, y) for y in ADDat

time = Numeric.arange( ADData.shape[0] )*1.0/ActualRate

pylab.plot(time, data_in_volts, 'o-')
pylab.xlabel('time (sec)')
pylab.ylabel('Volts')
pylab.savefig('example3.png',dpi=72)
pylab.show()
```

The output looks much better:



Example 4 - computing the power spectrum

Now we can use the function from pylab (part of matplotlib) to compute the power spectral density.

```
#example4.py
import UniversalLibrary as UL
import Numeric
import pylab

BoardNum = 0
Gain = UL.BIP5VOLTS

LowChan = 0
HighChan = 0

Count = 2000
Rate = 10000

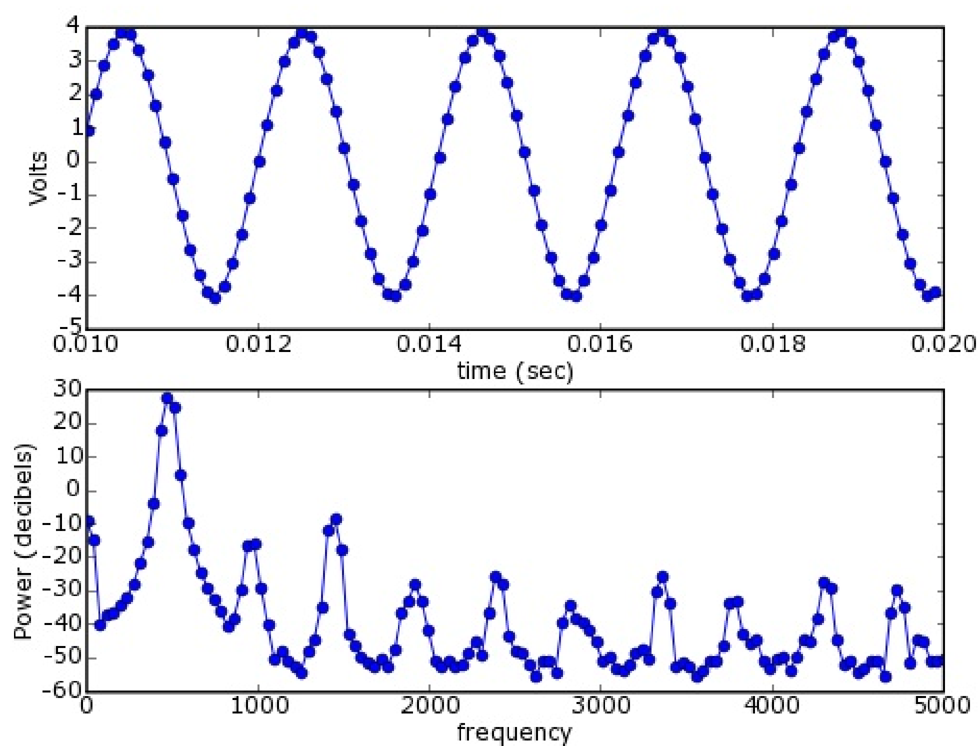
Options = UL.CONVERTDATA
ADData = Numeric.zeros((Count,), Numeric.Int16)

ActualRate = UL.cbAInScan(BoardNum, LowChan, HighChan, Count,
                     Rate, Gain, ADData, Options)
time = Numeric.arange( ADData.shape[0] )*1.0/ActualRate

# convert to Volts
data_in_volts = [ UL.cbToEngUnits(BoardNum, Gain, y) for y in ADDat
data_in_volts = Numeric.array(data_in_volts) # convert to Numeric a

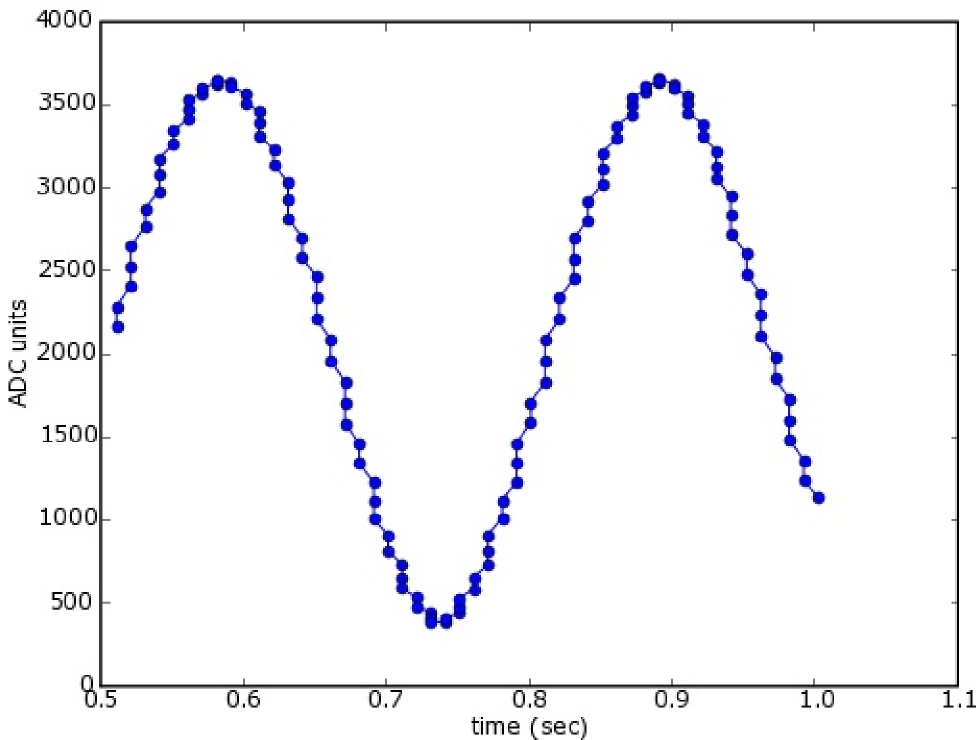
pxx, freqs = pylab.psd( data_in_volts, Fs=ActualRate )
decibels = 10*Numeric.log10(pxx)
pylab.subplot(2,1,1)
pylab.plot(time[100:200],data_in_volts[100:200],'o-') # plot a few
pylab.xlabel('time (sec)')
pylab.ylabel('Volts')
pylab.subplot(2,1,2)
pylab.plot(freqs, decibels, 'o-')
pylab.xlabel('frequency')
pylab.ylabel('Power (decibels)')
pylab.savefig('example4.png',dpi=72)
pylab.show()
```

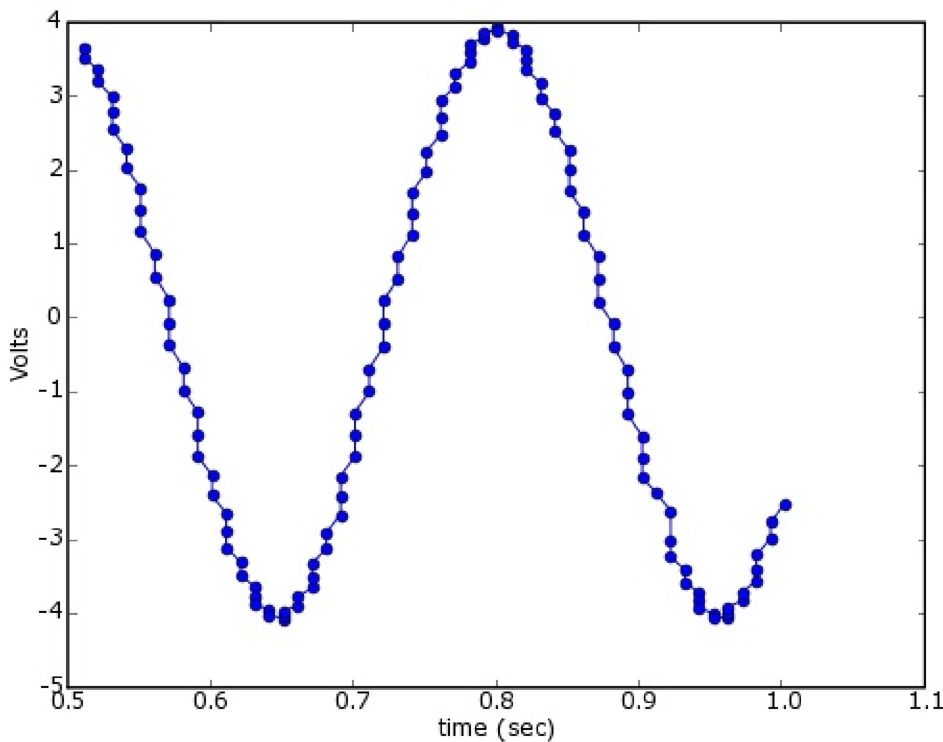
For this example, I’ve turned up the frequency on the function generator to 480 Hz. You can see, indeed, that’s what the `psd()` function tells us:

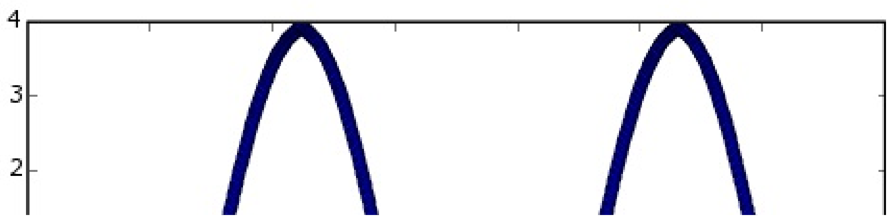


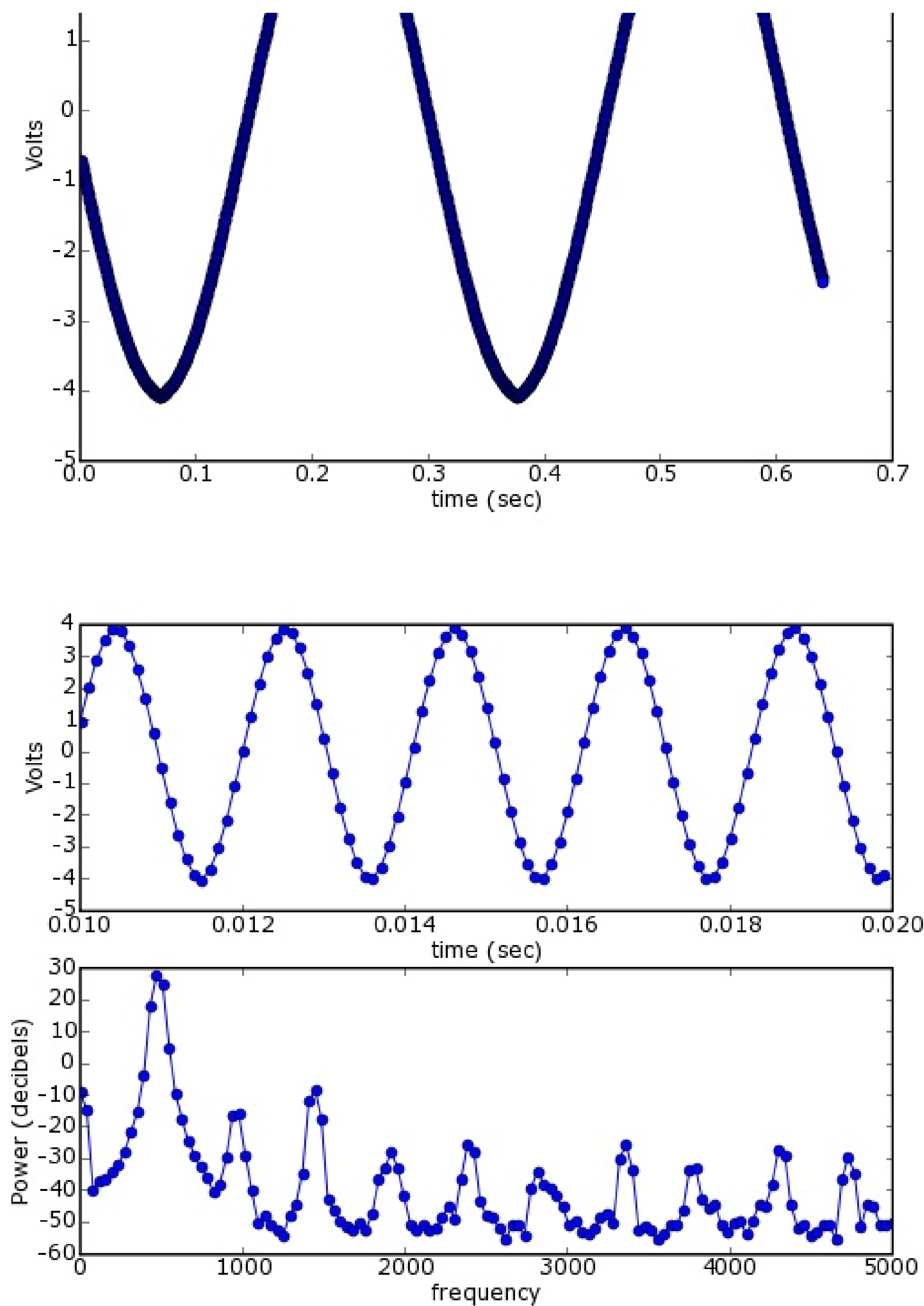
Attachments

- `example1.png`
- `example2.png`
- `example3.png`
- `example4.png`


4000
3500
3000
2500
2000
1500
1000
500
0
ADC units
0.5
0.6
0.7
0.8
0.9
1.0
1.1
time (sec)


4
3
2
1
0
-1
-2
-3
-4
-5
Volts
0.5
0.6
0.7
0.8
0.9
1.0
1.1
time (sec)


4
3
2


Volts
time (sec)
Volts
time (sec)
Power (decibels)
frequency

Fortran I/O Formats

> **NOTE**: you may want to use scipy.io.FortranFile instead.

Files written by Fortran programs can be written using one of two formats: formatted or unformatted. Formatted files are written in human-readable formats and it should be possible to load them using numpy.fromfile. Unformatted files are written using a binary format that is unspecified by the Fortran standard. In practice, most compilers/runtimes use a record-based format with an integer header consisting of the length of the record in bytes, then the record itself followed by an integer footer with the length of the preceeding in bytes.

Given that the precision and endian-ness of the headers and the data are unspecified, there are a large number of possible combinations that may be seen in the wild. The *FortranFile* class can deal with a great many of these.

The following is an example of how to read a particular unformatted output file. Note the presence of the 'i4' elements of the dtype representing the header and the footer.

Reading FORTRAN "unformatted IO" files

Lots of scientific code is written in FORTRAN. One of the most convenient file formats to create in FORTRAN is the so-called "unformatted binary file". These files have all the disadvantages of raw binary IO - no metadata, data depends on host endianness, floating-point representation, and possibly word size - but are not simply raw binary. They are organized into "records", which are padded with size information. Nevertheless, one does encounter such files from time to time. No prewritten code appears to be available to read them in numpy/scipy, but it can be done with relative ease using numpy's record arrays:

```
>>> A = N.fromfile("/tmp/tmp_i7j_a/resid2.tmp",
...   N.dtype([('pad1','i4'),
...    ('TOA','f8'),
...    ('resid_p','f8'),
...    ('resid_s','f8'),
...    ('orb_p','f8'),
...    ('f','f8'),
...    ('wt','f8'),
...    ('sig','f8'),
...    ('preres_s','f8'),
...    ('pad3','i8'),
...    ('pad2','i4')]))
```

This example is designed to read [http://www.atnf.csiro.au/research/pulsar/tempo/ref_man_sections/output.txt a file] output by [http://www.atnf.csiro.au/research/pulsar/tempo/ TEMPO]. Most of the fields, “TOA” up to “preres_s”, are fields that are present and of interest in the file. The field “pad3” is either an undocumented addition to the file format or some kind of padding (it is always zero in my test file). The FORTRAN unformatted I/O adds the fields “pad1” and “pad2”. Each should contain the length, in bytes, of each record (so the presence of the extra “pad3” field could be deduced). This code ignores t

Input and output

Introduction

This page gives examples how to read or write a !NumPy array to or from a file, be it ascii or binary. The various methods demonstrated all have copious and sometimes sophisticated options, call help to get details.

We will consider a trivial example where we create an array of zeros called `data`, write it to a file `myfile.txt` (myfile.dat for the binary case), and read it into `read_data`.

This documentation could be improved by discussing more sophisticated cases (e.g. multiple arrays), and discussing the costs/benefits of the various approaches presented.

Text files

SciPy

Writing files can be accomplished using `savetxt`. By far the easiest way to read text data is via genfromtxt, (or derivative convenience functions recfromtxt and recfromcsv).

```
>>> from numpy import *
>>> data = zeros((3,3))
>>>#Write data:
>>> savetxt("myfile.txt", data)
>>>#Read:
>>> data = genfromtxt("myfile.txt") }}}

== Matplotlib (pylab) ==

Matplotlib  provides an easy solution which seems to load data fast

{{{#!python numbers=disable
>>> from numpy import *
>>> from pylab import load           # warning, the load() function
>>> from pylab import save
>>> data = zeros((3,3))
>>> save('myfile.txt', data)
>>> read_data = load("myfile.txt")
```

numPy

```
>>> savetxt('myfile.txt', data, fmt="%12.6G")    # save to file
```

```
>>> from numpy import *
>>> data = genfromtxt('table.dat', unpack=True)
```

csv files

Note that csv stands for “comma separated value”. This means that the separator (also called a delimiter), i.e. the character which is used to separate individual values in a file, is a comma. In the examples above, the default delimiter is a space, but all of the above methods have an option (see their respective help for details), which can be set to a comma in order to read or write a csv file instead.

A more sophisticated example

Or, assuming you have imported numpy as N, you may want to read arbitrary column types. You can also return a recarray, which let’s you assign ‘column headings’ to your array.

```
def read_array(filename, dtype, separator=','):
    """ Read a file with an arbitrary number of columns.
 The type of data in each column is arbitrary
 It will be cast to the given dtype at runtime
 """
    cast = N.cast
    data = [[] for dummy in xrange(len(dtype))]
    for line in open(filename, 'r'):
        fields = line.strip().split(separator)
        for i, number in enumerate(fields):
            data[i].append(number)
    for i in xrange(len(dtype)):
        data[i] = cast[dtype[i]](data[i])
    return N.rec.array(data, dtype=dtype)
```

This can then be called with the corresponding dtype:

```
mydescr = N.dtype([('column1', 'int32'), ('column2Name', 'uint32'),
myrecarray = read_array('file.csv', mydescr)
```

Binary Files

The advantage of binary files is the huge reduction in file size. The price paid is losing human readability, and in some formats, losing portability.

Let us consider the array in the previous example.

File format with metadata

The simplest possibility is to use 's own binary file format. See , and .

```
>>> numpy.save('test.npy', data)
>>> data2 = numpy.load('test.npy')
```

You can save several arrays in a single file using . When loading an file you get an object of type . You can obtain a list of arrays and load individual arrays like this:

```
>>> numpy.savez('foo.npz', a=a,b=b)
>>> foo = numpy.load('foo.npz')
>>> foo.files
['a', 'b']
>>> a2 = foo['a']
>>> b2 = foo['b']
```

On older systems, the standard was to use python's pickle module to pickle the arrays.

Raw binary

These file formats simply write out the internal representation of the arrays. This is platform-dependent and includes no information about array shape or datatype, but is quick and easy.

SciPy provides fwrite() from scipy.io.numpyio. You have to set the size of your data, and optionally, its type (integer, short, float, etc; see 1).

For reading binary files, scipy.io.numpyio provides fread(). You have to know the datatype of your array, its size and its shape.

```
>>> from scipy.io.numpyio import fwrite, fread
>>> data = zeros((3,3))
>>>#write:  fd = open('myfile.dat', 'wb')
>>> fwrite(fd, data.size, data)
>>> fd.close()
>>>#read:
>>> fd = open('myfile.dat', 'rb')
>>> datatype = 'i'
>>> size = 9
>>> shape = (3,3)
>>> read_data = fread(fd, size, datatype)
>>> read_data = data.reshape(shape)
```

Or, you can simply use and . Following the previous example:

```
>>> data.tofile('myfile.dat')
>>> fd = open('myfile.dat', 'rb')
>>> read_data = numpy.fromfile(file=fd, dtype=numpy.uint8).reshape(
```

```
numpy data type. The option {{{fromfile(..., count=<number>)}}} spe
If you want that, use {{{numpy}}}'s own binary file format. See {{
{{{#! python numbers=disable
>>> numpy.save('test.npy', data)
>>> data2 = numpy.load('test.npy')
```

Another, but deprecated, way to fully control endianness (byteorder), storage order (row-major, column-major) for rank > 1 arrays and datatypes that are written and read back is . Writing:

```
>>> from scipy.io import npfile
>>> shape = (3,3)
>>> data = numpy.random.random(shape)
>>> npf = npfile('test.dat', order='F', endian='<', permission='wb
>>> npf.write_array(data)
>>> npf.close()
```

And reading back:

```
>>> npf = npfile('test.dat', order='F', endian='<', permission='rb
>>> data2 = npf.read_array(float, shape=shape)
>>> npf.close()
```

Write a Fortran or C array to a binary file with metadata

libnpy is a small library that provides simple routines for saving a C or Fortran array to a data file using NumPy's own binary format. For a description of this format, do

```
>>> from numpy.lib import format
>>> help(format)
```

Here is a minimal C example `cex.c` :

```
#include"npy.h"
int main(){
    double a[2][4] = { { 1, 2, 3, 4 },
                       { 5, 6, 7, 8 } };
    int shape[2] = { 2, 4 }, fortran_order = 0;

    npy_save_double("ca.npy", fortran_order, 2, shape, &a[0][0]);
    return 0;
}
```

The program creates a file `ca.npy` that you can load into python in the usual way.

```
>>> ca = np.load('ca.npy')
>>> print ca
[[ 1\.  2\.  3\.  4.]
 [ 5\.  6\.  7\.  8.]]
```

The corresponding Fortran program, `fex.f95` , looks like

```
program fex
    use fnpy
    use iso_c_binding
    implicit none

    integer  :: i
    real(C_DOUBLE) :: a(2,4) = reshape([(i, i=1,8)], [2,4])

    call save_double("fa.npy", shape(a), a)
end program fex
```

but the entries of the NumPy array now follow the Fortran (column-major) ordering.

```
>>> fa = np.load('fa.npy')
>>> print fa
[[ 1\.  3\.  5\.  7.]
 [ 2\.  4\.  6\.  8.]]
```

The `README` file in the source distribution explains how to compile the library using `make`.

If you put `npy.h` and `libnpy.a` in the same directory as `cex.c`, then you can build the executable `cex` with the command

```
gcc -o cex cex.c libnpy.a
```

Similarly, with `npy.mod` and `libnpy.a` in the same directory as `fex.f95`, build `fex` with the command

```
gfortran -o fex fex.f95 libnpy.a
```

LAS reader

This cookbook example contains a module that implements a reader for a LAS (Log ASCII Standard) well log file (LAS 2.0). See the Canadian Well Logging Society page about this format for more information.

```
#!python
"""LAS File Reader

The main class defined here is LASReader, a class that reads a LAS
and makes the data available as a Python object.
"""

# Copyright (c) 2011, Warren Weckesser
#
# Permission to use, copy, modify, and/or distribute this software
# purpose with or without fee is hereby granted, provided that the
# copyright notice and this permission notice appear in all copies.
#
# THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WAR
# WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIAB
# ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DA
# WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER I
# ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING
# OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

import re
import keyword

import numpy as np

def isidentifier(s):
    if s in keyword.kwlist:
        return False
    return re.match(r'^[a-z_][a-z0-9_]*$', s, re.I) is not None

def _convert_to_value(s):
    try:
        value = int(s)
    except ValueError:
        try:
            value = float(s)
        except ValueError:
            value = s
    return value

class LASError(Exception):
    pass
```

```
class LASItem(object):
    """This class is just a namespace, holding the attributes 'name
 'units', 'data', 'value', and 'descr'.  'value' is the numerical
 value of 'data', if it has a numerical value (specifically, if
 int() or float() don't raise an exception when given the value
 of the 'data' attribute).

 A class method, from_line(cls, line), is provided to parse
 a line from a LAS file and create a LASItem instance.
 """
    def __init__(self, name, units='', data='', descr=''):
        self.name = name
        self.units = units
        self.data = data
        self.value = _convert_to_value(data)
        self.descr = descr

    def __str__(self):
        s = ("name='%s', units='%s', data='%s', descr='%s'" %
                (self.name, self.units, self.data, self.descr))
        return s

    def __repr__(self):
        s = str(self)
        return "LASItem(%s)" % s

    @classmethod
    def from_line(cls, line):
        first, descr = line.rsplit(':', 1)
        descr = descr.strip()
        name, mid = first.split('.', 1)
        name = name.strip()
        if mid.startswith(' '):
            # No units
            units = ''
            data = mid
        else:
            units_data = mid.split(None, 1)
            if len(units_data) == 1:
                units = units_data[0]
                data = ''
            else:
                units, data = units_data
        return LASItem(name=name, units=units, data=data.strip(),
                       descr=descr.strip())

def _read_wrapped_row(f, n):
    """Read a "row" of data from the Ascii section of a "wrapped" L

  `f` must be a file object opened for reading.
  `n` is the number of fields in the row.
```

```
  Returns the list of floats read from the file.
  """
     depth = float(f.readline().strip())
     values = [depth]
     while len(values) < n:
         new_values = [float(s) for s in f.readline().split()]
         values.extend(new_values)
     return values

def _read_wrapped_data(f, dt):
     data = []
     ncols = len(dt.names)
     while True:
         try:
             row = _read_wrapped_row(f, ncols)
         except Exception:
             break
         data.append(tuple(row))
     data = np.array(data, dtype=dt)
     return data

class LASSection(object):
     """Represents a "section" of a LAS file.

  A section is basically a collection of items, where each item has
  attributes 'name', 'units', 'data' and 'descr'.

  Any item in the section whose name is a valid Python identifier is
  also attached to the object as an attribute.  For example, if `s`
  LASSection instance, and the corresponding section in the LAS file
  contained this line:

  FD   .K/M3               999.9999        :  Fluid Density

  then the item may be referred to as `s.FD` (in addition to the lon
  `s.items['FD']`).

  Attributes
  ----------
  items : dict
  The keys are the item names, and the values are LASItem instances.
  names : list
  List of item names, in the order they were read from the LAS file.

  """
     def __init__(self):
         # Note: In Python 2.7, 'items' could be an OrderedDict, and
         # then 'names' would not be necessary--one could use items.
         self.items = dict()
         self.names = []

     def add_item(self, item):
         self.items[item.name] = item
```

```
        self.names.append(item.name)
        if isidentifier(item.name) and not hasattr(self, item.name)
            setattr(self, item.name, item)

    def display(self):
        for name in self.names:
            item = self.items[name]
            namestr = name
            if item.units != '':
                namestr = namestr + (" (%s)" % item.units)
            print "%-16s  %-30s [%s]" % (namestr, "'" + item.data +
                                        item.descr)

class LASReader(object):
    """The LASReader class holds data from a LAS file.

 This reader only handles LAS 2.0 files (as far as I know).

 Constructor
 -----------
 LASReader(f, null_subs=None)

 f : file object or string
 If f is a file object, it must be opened for reading.
 If f is a string, it must be the filename of a LAS file.
 In that case, the file will be opened and read.

 Attributes for LAS Sections
 ---------------------------
 version : LASSection instance
 This LASSection holds the items from the '~V' section.

 well : LASSection instance
 This LASSection holds the items from the '~W' section.

 curves : LASection instance
 This LASSection holds the items from the '~C' section.

 parameters : LASSection instance
 This LASSection holds the items from the '~P' section.

 other : str
 Holds the contents of the '~O' section as a single string.

 data : numpy 1D structured array
 The numerical data from the '~A' section.  The data type
 of the array is constructed from the items in the '~C'
 section.

 Other attributes
 ----------------
 data2d : numpy 2D array of floats
 The numerical data from the '~A' section, as a 2D array.
```

```
This is a view of the same data as in the `data` attribute.

wrap : bool
True if the LAS file was wrapped. (More specifically, this
attribute is True if the data field of the item with the
name 'WRAP' in the '~V' section has the value 'YES'.)

vers : str
The LAS version. (More specifically, the value of the data
field of the item with the name 'VERS' in the '~V' section).

null : float or None
The numerical value of the 'NULL' item in the '~W' section.
The value will be None if the 'NULL' item was missing.

null_subs : float or None
The value given in the constructor, to be used as the
replacement value of each occurrence of `null_value` in
the log data.  The value will be None (and no substitution
will be done) if the `null_subs` argument is not given to
the constructor.

start : float, or None
Numerical value of the 'STRT' item from the '~W' section.
The value will be None if 'STRT' was not given in the file.

start_units : str
Units of the 'STRT' item from the '~W' section.
The value will be None if 'STRT' was not given in the file.

stop : float
Numerical value of the 'STOP' item from the '~W' section.
The value will be None if 'STOP' was not given in the file.

stop_units : str
Units of the 'STOP' item from the '~W' section.
The value will be None if 'STOP' was not given in the file.

step : float
Numerical value of the 'STEP' item from the '~W' section.
The value will be None if 'STEP' was not given in the file.

step_units : str
Units of the 'STEP' item from the '~W' section.
The value will be None if 'STEP' was not given in the file.

"""

   def __init__(self, f, null_subs=None):
       """f can be a filename (str) or a file object.

If 'null_subs' is not None, its value replaces any values in the d
that matches the NULL value specified in the Version section of th
```

```
file.
"""
        self.null = None
        self.null_subs = null_subs
        self.start = None
        self.start_units = None
        self.stop = None
        self.stop_units = None
        self.step = None
        self.step_units = None

        self.version = LASSection()
        self.well = LASSection()
        self.curves = LASSection()
        self.parameters = LASSection()
        self.other = ''
        self.data = None

        self._read_las(f)

        self.data2d = self.data.view(float).reshape(-1, len(self.cu
        if null_subs is not None:
            self.data2d[self.data2d == self.null] = null_subs

    def _read_las(self, f):
        """Read a LAS file.

Returns a dictionary with keys 'V', 'W', 'C', 'P', 'O' and 'A',
corresponding to the sections of a LAS file.  The values associate
with keys 'V', 'W', 'C' and 'P' will be lists of Item instances.
value associated with the 'O' key is a list of strings.  The value
associated with the 'A' key is a numpy structured array containing
log data.  The field names of the array are the mnemonics from the
Curve section of the file.
"""
        opened_here = False
        if isinstance(f, basestring):
            opened_here = True
            f = open(f, 'r')

        self.wrap = False

        line = f.readline()
        current_section = None
        current_section_label = ''
        while not line.startswith('~A'):
            if not line.startswith('#'):
                if line.startswith('~'):
                    if len(line) < 2:
                        raise LASError("Missing section character a
                    current_section_label = line[1:2]
                    other = False
                    if current_section_label == 'V':
```

```
                    current_section = self.version
                elif current_section_label == 'W':
                    current_section = self.well
                elif current_section_label == 'C':
                    current_section = self.curves
                elif current_section_label == 'P':
                    current_section = self.parameters
                elif current_section_label == 'O':
                    current_section = self.other
                    other = True
                else:
                    raise LASError("Unknown section '%s'" % lin
            elif current_section is None:
                raise LASError("Missing first section.")
            else:
                if other:
                    # The 'Other' section is just lines of text
                    # assemble them into a single string.
                    self.other += line
                    current_section = self.other
                else:
                    # Parse the line into a LASItem and add it
                    # current section.
                    m = LASItem.from_line(line)
                    current_section.add_item(m)
                    # Check for the required items whose values
                    # store as attributes of the LASReader inst
                    if current_section == self.version:
                        if m.name == 'WRAP':
                            if m.data.strip() == 'YES':
                                self.wrap = True
                        if m.name == 'VERS':
                            self.vers = m.data.strip()
                    if current_section == self.well:
                        if m.name == 'NULL':
                            self.null = float(m.data)
                        elif m.name == 'STRT':
                            self.start = float(m.data)
                            self.start_units = m.units
                        elif m.name == 'STOP':
                            self.stop = float(m.data)
                            self.stop_units = m.units
                        elif m.name == 'STEP':
                            self.step = float(m.data)
                            self.step_units = m.units
        line = f.readline()

    # Finished reading the header--all that is left is the nume
    # data that follows the '~A' line.  We'll construct a struc
    # data type, and, if the data is not wrapped, use numpy.loa
    # to read the data into an array.  For wrapped rows, we use
    # function _read_wrapped() defined elsewhere in this module
    # The data type is determined by the items from the '~Curve
```

```
        dt = np.dtype([(name, float) for name in self.curves.names]
        if self.wrap:
            a = _read_wrapped_data(f, dt)
        else:
            a = np.loadtxt(f, dtype=dt)
        self.data = a

        if opened_here:
            f.close()

if __name__ == "__main__":
    import sys

    las = LASReader(sys.argv[1], null_subs=np.nan)
    print "wrap? ", las.wrap
    print "vers? ", las.vers
    print "null =", las.null
    print "start =", las.start
    print "stop  =", las.stop
    print "step  =", las.step
    print "Version ---"
    las.version.display()
    print "Well ---"
    las.well.display()
    print "Curves ---"
    las.curves.display()
    print "Parameters ---"
    las.parameters.display()
    print "Other ---"
    print las.other
    print "Data ---"
    print las.data2d
```

Source code: `las.py`

Here's an example of the use of this module:

```
>>> import numpy as np
>>> from las import LASReader
>>> sample3 = LASReader('sample3.las', null_subs=np.nan)
>>> print sample3.null
-999.25
>>> print sample3.start, sample3.stop, sample3.step
910.0 909.5 -0.125
>>> print sample3.well.PROV.data, sample3.well.UWI.data
ALBERTA 100123401234W500
>>> from matplotlib.pyplot import plot, show
>>> plot(sample3.data['DEPT'], sample3.data['PHIE'])
[<matplotlib.lines.Line2D object at 0x4c2ae90>]
>>> show()
```

It creates the following plot:

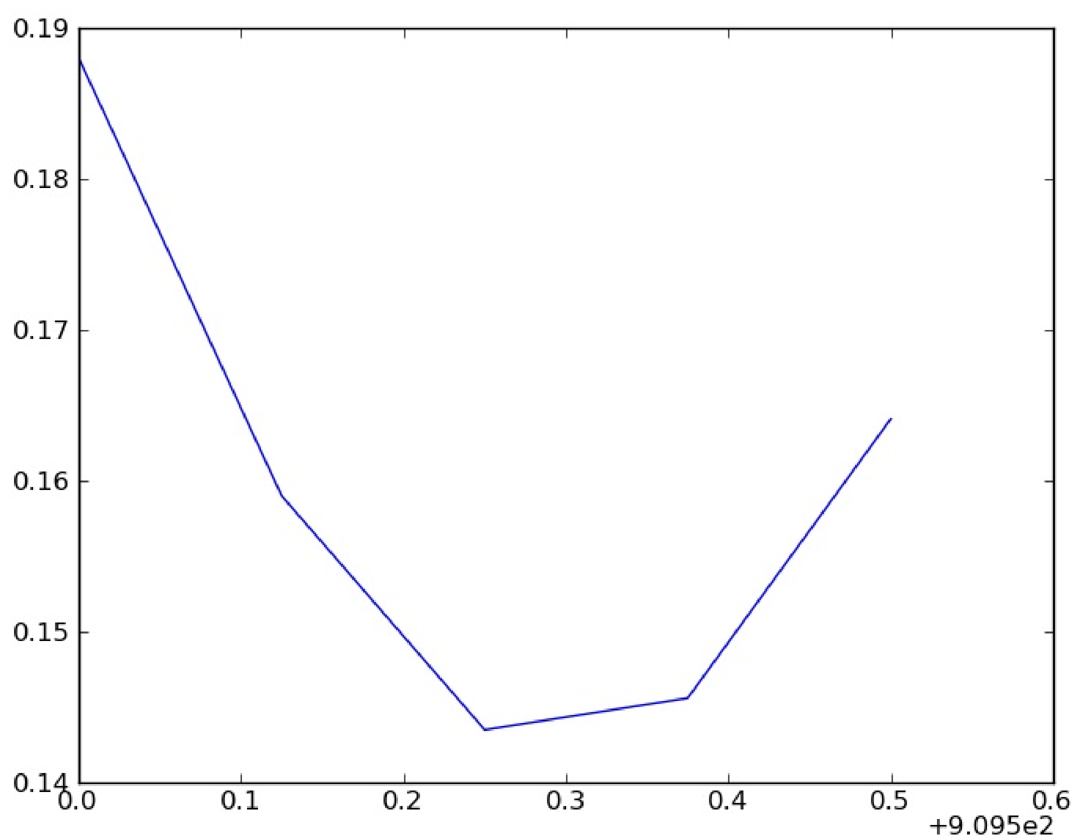
`sample3plot.png`

The sample LAS file is here:

`sample3.las`

Attachments

- `las.py`
- `sample3.las`
- `sample3plot.png`



Reading SPE file from CCD camera

Some charge-coupled device (CCD) cameras (Princeton and like) produce SPE files. This page suggests how to read such binary files with Numpy but the code is not robust. The following code is only able to read files having the same format as the example, 'lampe_dt.spe' (unfortuanetly the only SPE file on the wiki).

Loading SPE file with numpy

Only Numpy is required for loading SPE file, the result will be an array made of colors. The image size is at position 42 and 656 and the data at 4100. There are then many other data in a SPE file header, one must be the data type (you are welcome to edit this page if you know where). Finally note that the image is always made of colors coded on unsigned integer of 16 bits but it might not be the case in your input file.

```
#!python numbers=disabled
# read_spe.py
import numpy as N

class File(object):

    def __init__(self, fname):
        self._fid = open(fname, 'rb')
        self._load_size()

    def _load_size(self):
        self._xdim = N.int64(self.read_at(42, 1, N.int16)[0])
        self._ydim = N.int64(self.read_at(656, 1, N.int16)[0])

    def _load_date_time(self):
        rawdate = self.read_at(20, 9, N.int8)
        rawtime = self.read_at(172, 6, N.int8)
        strdate = ''
        for ch in rawdate :
            strdate += chr(ch)
        for ch in rawtime:
            strdate += chr(ch)
        self._date_time = time.strptime(strdate,"%d%b%Y%H%M%S")

    def get_size(self):
        return (self._xdim, self._ydim)

    def read_at(self, pos, size, ntype):
        self._fid.seek(pos)
        return N.fromfile(self._fid, ntype, size)

    def load_img(self):
        img = self.read_at(4100, self._xdim * self._ydim, N.uint16
        return img.reshape((self._ydim, self._xdim))

    def close(self):
        self._fid.close()

def load(fname):
    fid = File(fname)
    img = fid.load_img()
    fid.close()
    return img

if __name__ == "__main__":
    import sys
    img = load(sys.argv[-1])
```

Viewing the image with matplotlib and ipython

The ‘read_spe.py’ script from above and the ‘lampe_dt.spe’ example are provided in the archive `read_spe.zip` . Once decompresesed,)# you can then start ipython in the directory where the script lives:

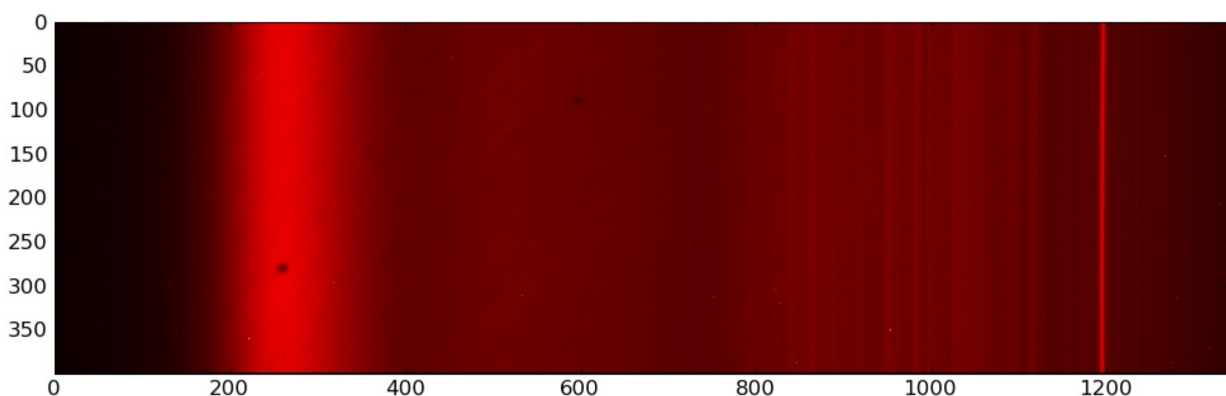
```
ipython -pylab read_spe.py lampe_dt.spe
```

The following first line will show the image in a new window. The second line will change the colormap (try ‘help(pylab.colormaps)’ for listing them).

```
#!python
>>> pylab.imshow(img)
>>> pylab.hot()
```

Attachments

- `lampe_dt.png`
- `read_spe.zip`



Reading mat files

```
Here are exemples of how to read two variables {{{lat}}} and {{{lon

= Matlab up to 7.1 =
mat files created with Matlab up to version 7.1 can be read using t

{{{
#!python
#!/usr/bin/env python
from scipy.io import loadmat
x = loadmat('test.mat')
lon = x['lon']
lat = x['lat']
# one-liner to read a single variable
lon = loadmat('test.mat')['lon']
```

hdf5 in Matlab

Python can save rich hierarchical datasets in hdf5 format. Matlab can read hdf5, but the api is so heavy it is almost unusable. Here are some matlab scripts (written by Gaël Varoquaux) to load and save data in hdf5 format under Matlab with the same signature as the standard matlab load/save function.

`hdf5matlab.zip`

These Matlab scripts cannot load every type allowed in hdf5. Feel free to provide python scripts to use pytables to implement simple load/save functions compatible with this hdf5 subset.

One notice: these script use the "Workspace" namespace to store some variables, they will pollute your workspace when saving data from Matlab. Nothing that I find unacceptable.

Another loader script

Here is a second HDF5 loader script, which loads (optionally partial) data from a HDF5 file to a Matlab structure

`h5load.m`

It can deal with more varied HDF5 datasets than the Matlab high-level functions (at least R2008a hdf5info fails with chunked compressed datasets), via using only the low-level HDF5 API.

The script also recognizes complex numbers in the Pytables format, and permutes array dimensions to match the logical order in the file (ie. to match Python. The builtin Matlab functions by default return data in the opposite order, so the first dimension in Python would be the last in Matlab).

Attachments

- `h5load.m`
- `hdf5matlab.zip`

Matplotlib / 3D Plotting

Matplotlib VTK integration

Just in case you would ever like to incorporate matplotlib plots into your vtk application, vtk provides a really easy way to import them.

Here is a full example for now:

```
from vtk import *

import matplotlib
matplotlib.use('Agg')
from matplotlib.figure import Figure
from matplotlib.backends.backend_agg import FigureCanvasAgg
import pylab as p

# The vtkImageImporter will treat a python string as a void pointer
importer = vtkImageImport()
importer.SetDataScalarTypeToUnsignedChar()
importer.SetNumberOfScalarComponents(4)

# It's upside-down when loaded, so add a flip filter
imflip = vtkImageFlip()
imflip.SetInput(importer.GetOutput())
imflip.SetFilteredAxis(1)

# Map the plot as a texture on a cube
cube = vtkCubeSource()

cubeMapper = vtkPolyDataMapper()
cubeMapper.SetInput(cube.GetOutput())

cubeActor = vtkActor()
cubeActor.SetMapper(cubeMapper)

# Create a texture based off of the image
cubeTexture = vtkTexture()
cubeTexture.InterpolateOn()
cubeTexture.SetInput(imflip.GetOutput())
cubeActor.SetTexture(cubeTexture)

ren = vtkRenderer()
ren.AddActor(cubeActor)

renWin = vtkRenderWindow()
renWin.AddRenderer(ren)

iren = vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)
```

```
# Now create our plot
fig = Figure()
canvas = FigureCanvasAgg(fig)
ax = fig.add_subplot(111)
ax.grid(True)
ax.set_xlabel('Hello from VTK!', size=16)
ax.bar(xrange(10), p.rand(10))

# Powers of 2 image to be clean
w,h = 1024, 1024
dpi = canvas.figure.get_dpi()
fig.set_figsize_inches(w / dpi, h / dpi)
canvas.draw() # force a draw

# This is where we tell the image importer about the mpl image
extent = (0, w - 1, 0, h - 1, 0, 0)
importer.SetWholeExtent(extent)
importer.SetDataExtent(extent)
importer.SetImportVoidPointer(canvas.buffer_rgba(0,0), 1)
importer.Update()

iren.Initialize()
iren.Start()
```

To have the plot be a billboard:

```
bbmap = vtkImageMapper()
bbmap.SetColorWindow(255.5)
bbmap.SetColorLevel(127.5)
bbmap.SetInput(imflip.GetOutput())

bbact = vtkActor2D()
bbact.SetMapper(hmap)
```

Comments

```
From zunzun Fri Aug 19 07:06:44 -0500 2005
From: zunzun
Date: Fri, 19 Aug 2005 07:06:44 -0500
Subject:
Message-ID: <20050819070644-0500@www.scipy.org>

from http://sourceforge.net/mailarchive/forum.php?thread_id=788446

If pylab is imported before vtk, everything works fine:

import pylab, vtkpython
pylab.ylabel("Frequency\n", multialignment="center", rotation=90)
n, bins, patches = pylab.hist([1,1,1,2,2,3,4,5,5,5,8,8,8,8], 5)
pylab.show()

If however vtk is imported first:

import vtkpython, pylab
pylab.ylabel("Frequency\n", multialignment="center", rotation=90)
n, bins, patches = pylab.hist([1,1,1,2,2,3,4,5,5,5,8,8,8,8], 5)
pylab.show()

then the Y axis label is positioned incorrectly on the plots.
```

```
From earthman Tue Oct 25 15:21:14 -0500 2005
From: earthman
Date: Tue, 25 Oct 2005 15:21:14 -0500
Subject:
Message-ID: <20051025152114-0500@www.scipy.org>

The reason for this is that vtk comes with it's own freetype librar
```

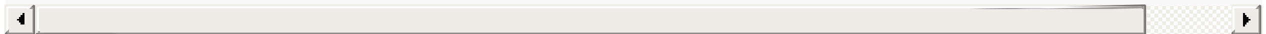
```
From mroublic Tue Jan 10 11:26:45 -0600 2006
From: mroublic
Date: Tue, 10 Jan 2006 11:26:45 -0600
Subject: One more change I had to make
Message-ID: <20060110112645-0600@www.scipy.org>
In-reply-to: <20050819070644-0500@www.scipy.org>

When I first tried this, I had the error:

Traceback (most recent call last):
  File "MatplotlibToVTK.py", line 61, in ?
    importer.SetImportVoidPointer(canvas.buffer_rgba(), 1)
TypeError: buffer_rgba() takes exactly 3 arguments (1 given)

I had to add 0,0 to the import line:
 importer.SetImportVoidPointer(canvas.buffer_rgba(0,0), 1)

I'm using VTK from CVS using the 5_0 Branch from around November 20
```



The above code didn’t run on my system. I had to change the following line: fig.set_figsize_inches(w / dpi, h / dpi) into: fig.set_figsize_inches(1.0_w / dpi, 1.0_h / dpi)

Matplotlib: mplot3d

The examples below show simple 3D plots using matplotlib. matplotlib's 3D capabilities were added by incorporating John Porter's mplot3d module, thus no additional download is required any more, the following examples will run with an up to date matplotlib installation. '''Note, this code is not supported in the matplotlib-0.98 branch, but you can use either the latest 0.99 release or the 0.91 maintenance version if you need this functionality. ''' Alternatively, the Mayavi2 project provides a pylab-like API for extensive 3D plotting: http://code.enthought.com/projects/mayavi/docs/development/html/mayavi/mlab.html

Note that not all examples on this page are up to date, so some of them might not be working. For other examples, see http://matplotlib.sourceforge.net/examples/mplot3d/

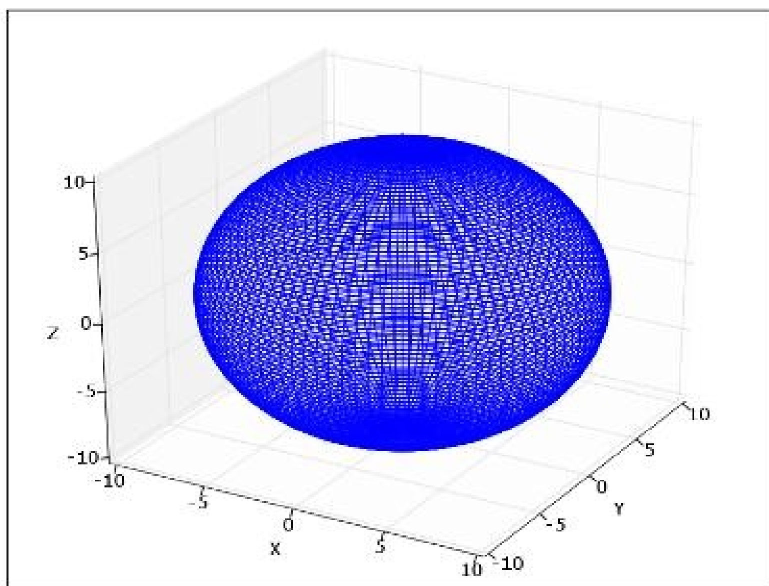
3D Plotting examples:

```
#!python
from numpy import *
import pylab as p
#import matplotlib.axes3d as p3
import mpl_toolkits.mplot3d.axes3d as p3

# u and v are parametric variables.
# u is an array from 0 to 2*pi, with 100 elements
u=r_[0:2*pi:100j]
# v is an array from 0 to 2*pi, with 100 elements
v=r_[0:pi:100j]
# x, y, and z are the coordinates of the points for plotting
# each is arranged in a 100x100 array
x=10*outer(cos(u),sin(v))
y=10*outer(sin(u),sin(v))
z=10*outer(ones(size(u)),cos(v))
```

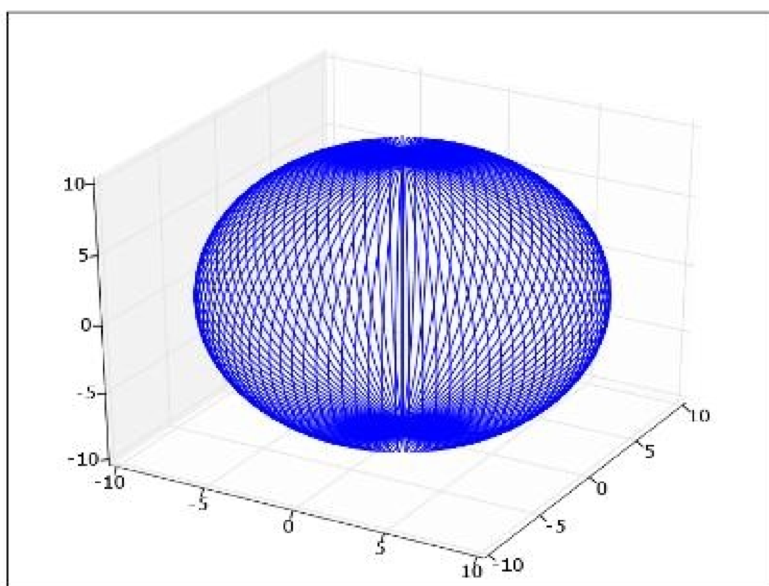
Wireframe (works on 0.87.5):

```
#!python
fig=p.figure()
ax = p3.Axes3D(fig)
ax.plot_wireframe(x,y,z)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
p.show()
```



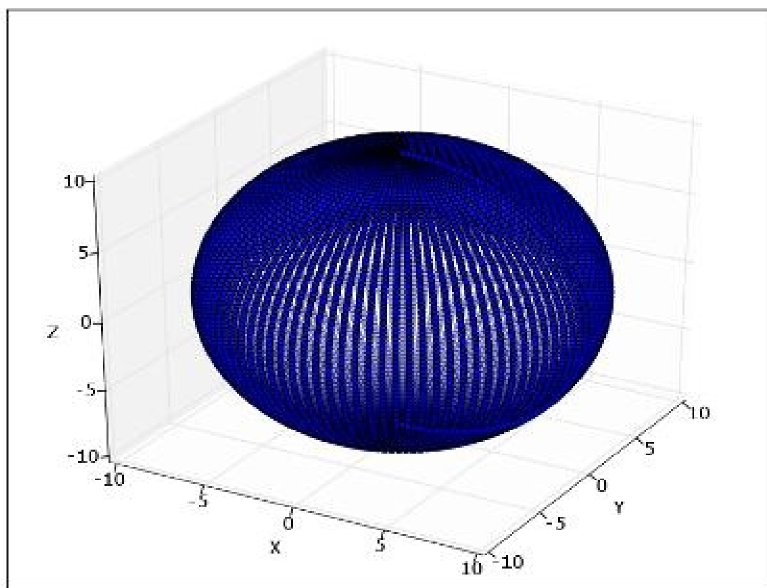
3D Plot:

```
#!python
# this connects each of the points with lines
fig=p.figure()
ax = p3.Axes3D(fig)
# plot3D requires a 1D array for x, y, and z
# ravel() converts the 100x100 array into a 1x10000 array
ax.plot3D(ravel(x),ravel(y),ravel(z))
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
fig.add_axes(ax)
p.show()
```



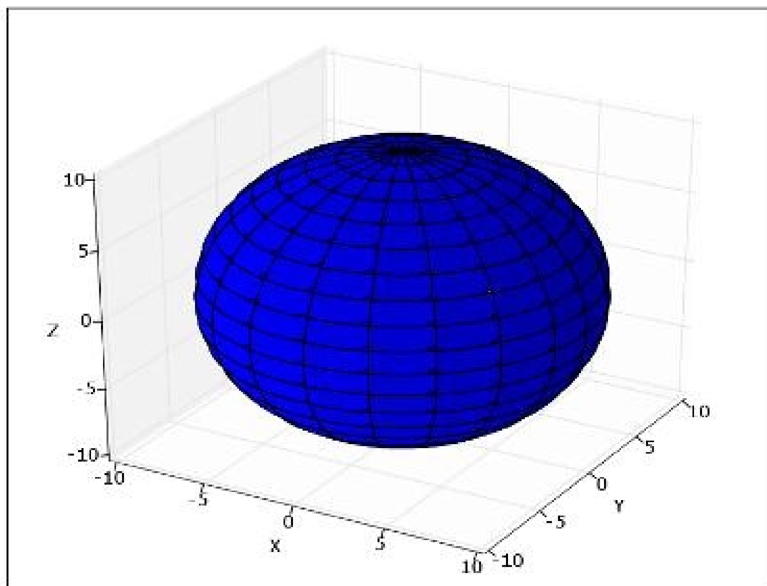
Scatter (works on 0.87.5, shows some artefacts):

```
#!python
fig=p.figure()
ax = p3.Axes3D(fig)
# scatter3D requires a 1D array for x, y, and z
# ravel() converts the 100x100 array into a 1x10000 array
ax.scatter3D(ravel(x),ravel(y),ravel(z))
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
p.show()
```



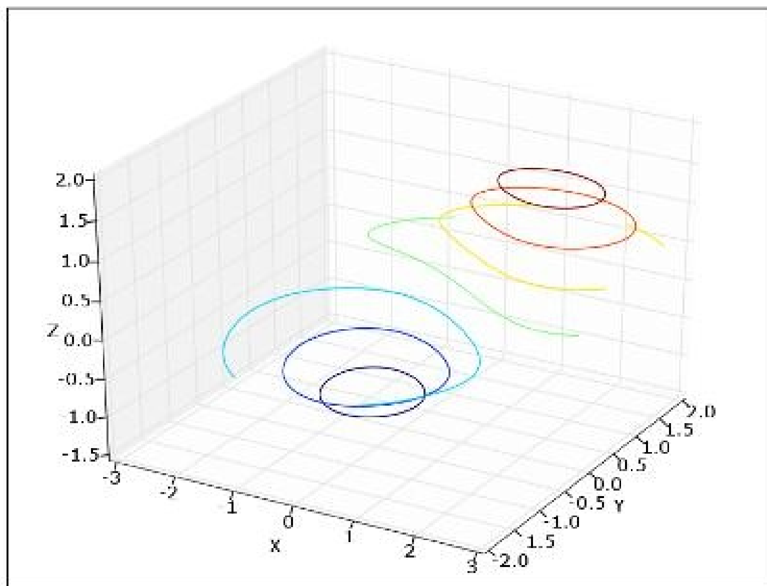
Surface (works on 0.87.5):

```
#!python
fig=p.figure()
ax = p3.Axes3D(fig)
# x, y, and z are 100x100 arrays
ax.plot_surface(x,y,z)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
p.show()
```



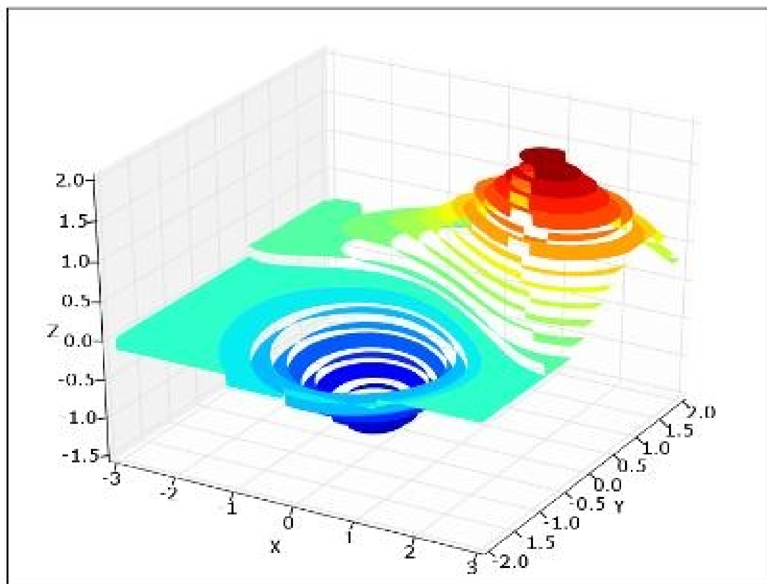
Contour3D (works on 0.87.5):

```
#!python
delta = 0.025
x = arange(-3.0, 3.0, delta)
y = arange(-2.0, 2.0, delta)
X, Y = p.meshgrid(x, y)
Z1 = p.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = p.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
# difference of Gaussians
Z = 10.0 * (Z2 - Z1)
fig=p.figure()
ax = p3.Axes3D(fig)
ax.contour3D(X,Y,Z)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
p.show()
```



Contourf3D:

```
#!python
# in mplt3D change:
# levels, colls = self.contourf(X, Y, Z, 20)
# to:
# C = self.contourf(X, Y, Z, *args, **kwargs)
# levels, colls = (C.levels, C.collections)
fig=p.figure()
ax = p3.Axes3D(fig)
ax.contourf3D(X,Y,Z)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
fig.add_axes(ax)
p.show()
```



2D Contour Plots (work on 0.87.5):

```
#!python
x=r_[-10:10:100j]
y=r_[-10:10:100j]
z= add.outer(x*x, y*y)
### Contour plot of z = x**2 + y**2
p.contour(x,y,z)
### ContourF plot of z = x**2 + y**2
p.figure()
p.contourf(x,y,z)
p.show()
```

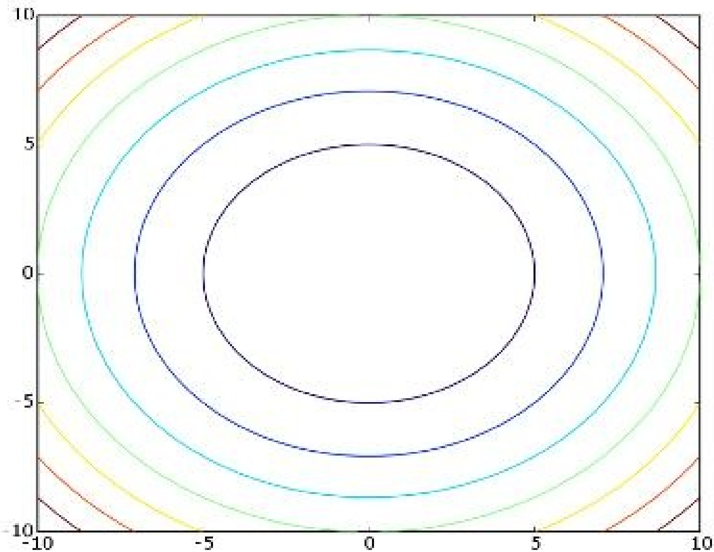


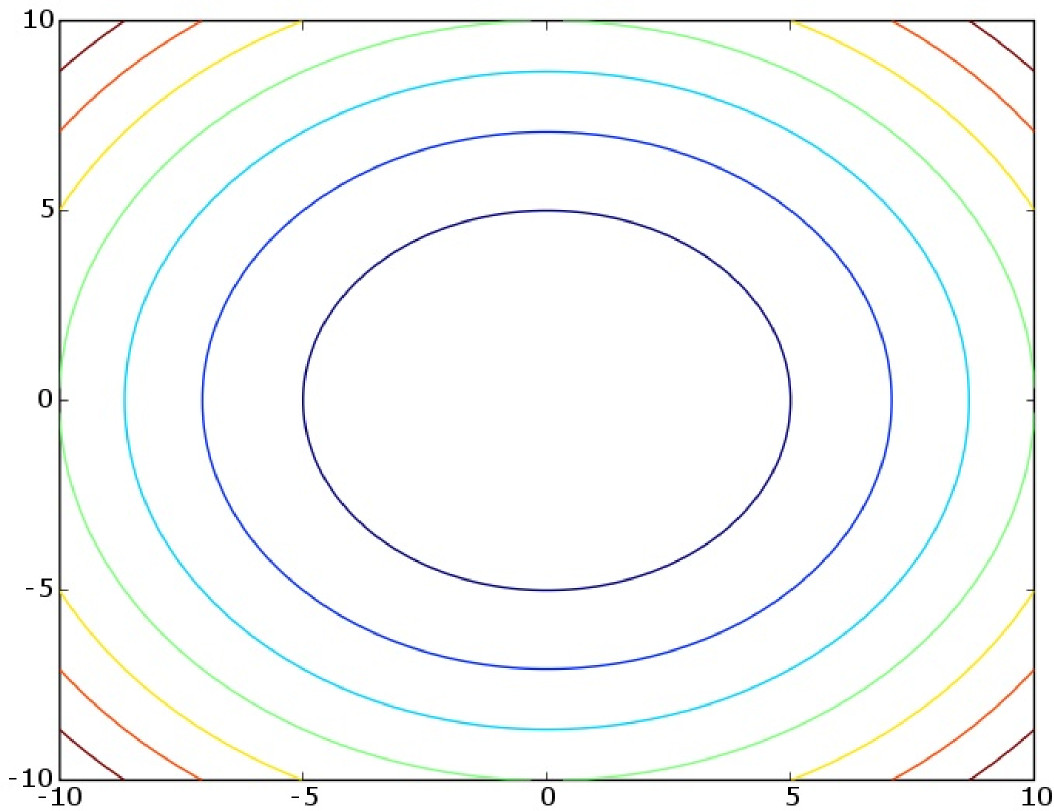
For some other examples of 3d plotting capability, run the following commands. See the source of matplotlib/axes3d.py for more information:

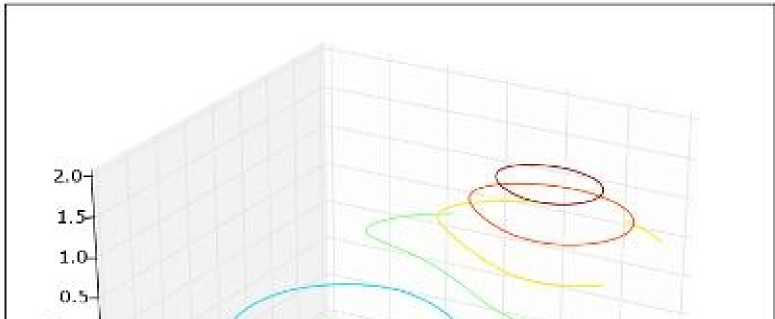
```
#!python
# note that for the following to work you have to modify the test f
#def test_xxxx():
#    import pylab
#    ax = Axes3D(pylab.figure())
#    ....
#    ....
#    pylab.show()
# the following then work on 0.87.5
p3.test_bar2D()
p3.test_contour()
p3.test_scatter()
p3.test_scatter2D()
p3.test_surface()
# the following fail on 0.87.5
p3.test_plot()
p3.test_polys()
p3.test_wir
```

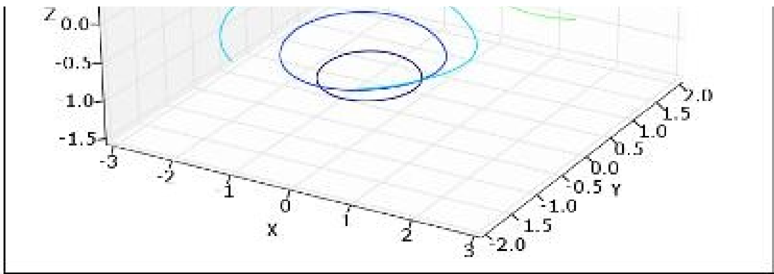
Attachments

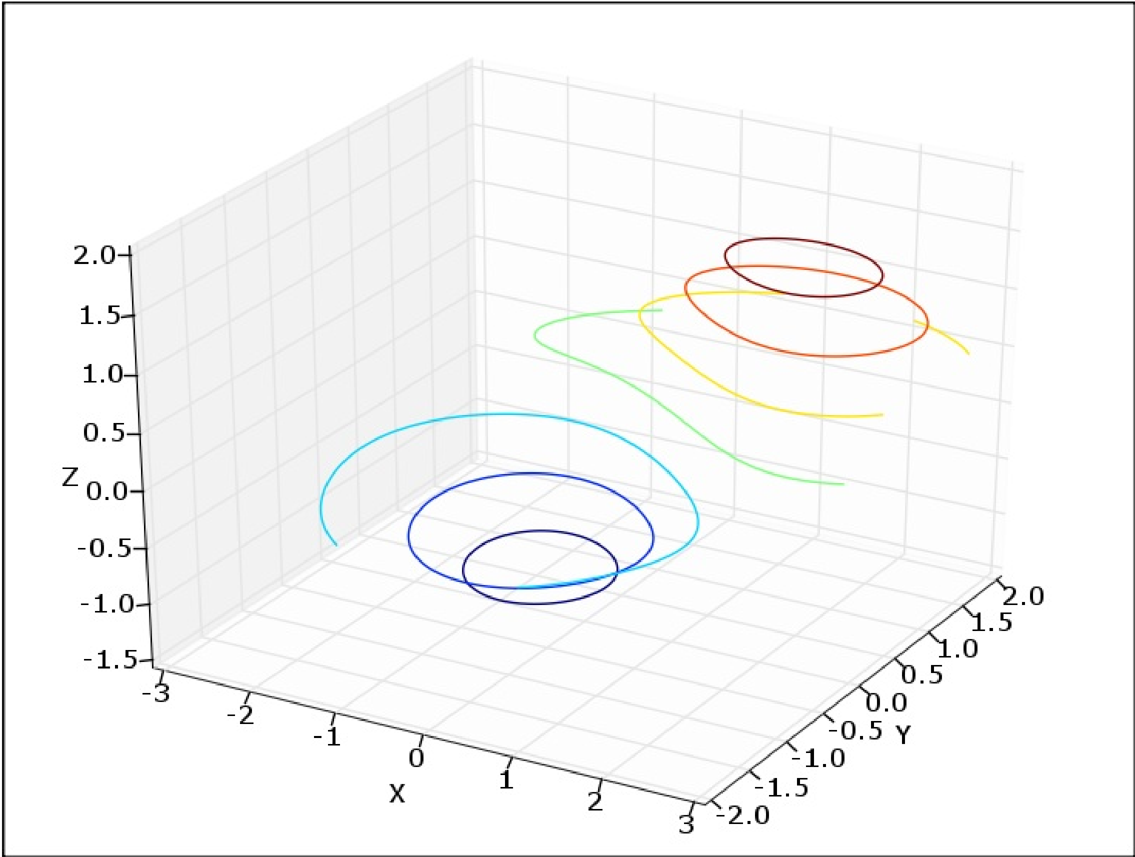
- `contour.jpg`
- `contour.png`
- `contour3D.jpg`
- `contour3D.png`
- `contourf.jpg`
- `contourf.png`
- `contourf3D.jpg`
- `contourf3D.png`
- `plot.jpg`
- `plot.png`
- `scatter.jpg`
- `scatter.png`
- `surface.jpg`
- `surface.png`
- `test1.jpg`
- `test1.png`
- `test2.jpg`
- `test2.png`
- `test3.jpg`
- `test3.png`
- `wireframe.jpg`
- `wireframe.png`

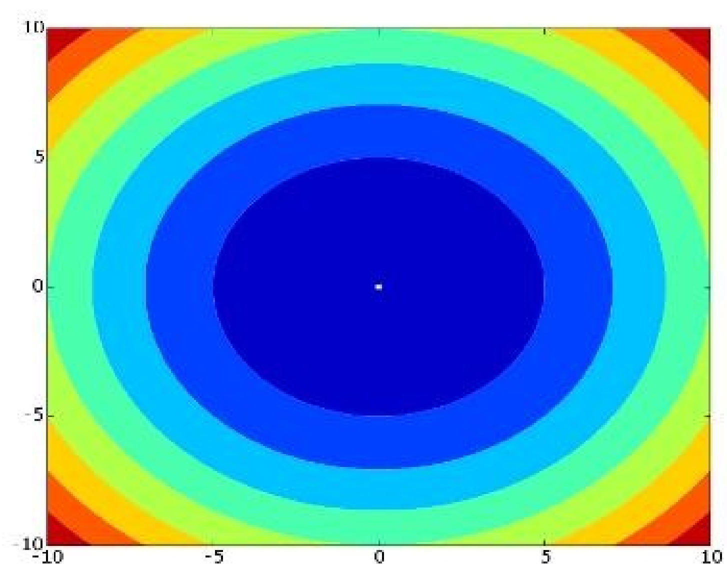

10
5
0
-5
-10
-10
-5
0
5
10

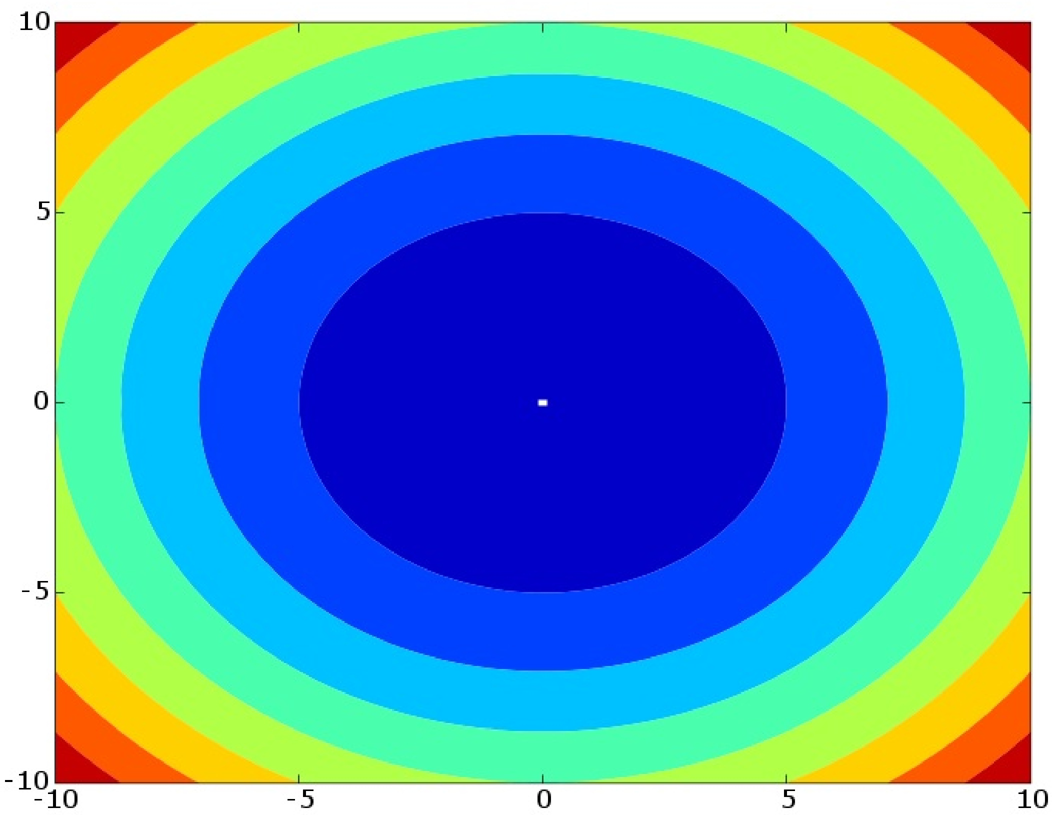

10
5
0
-5
-10
-10
-5
0
5
10

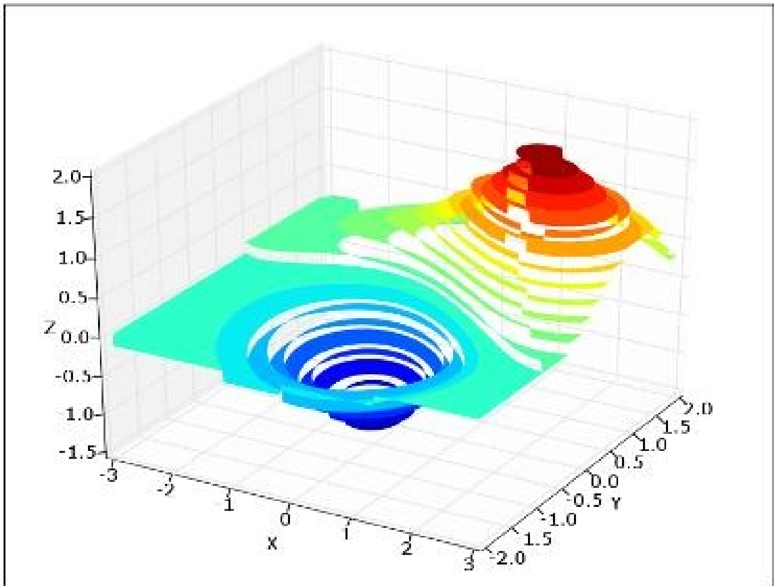

2.0
1.5
1.0
0.5

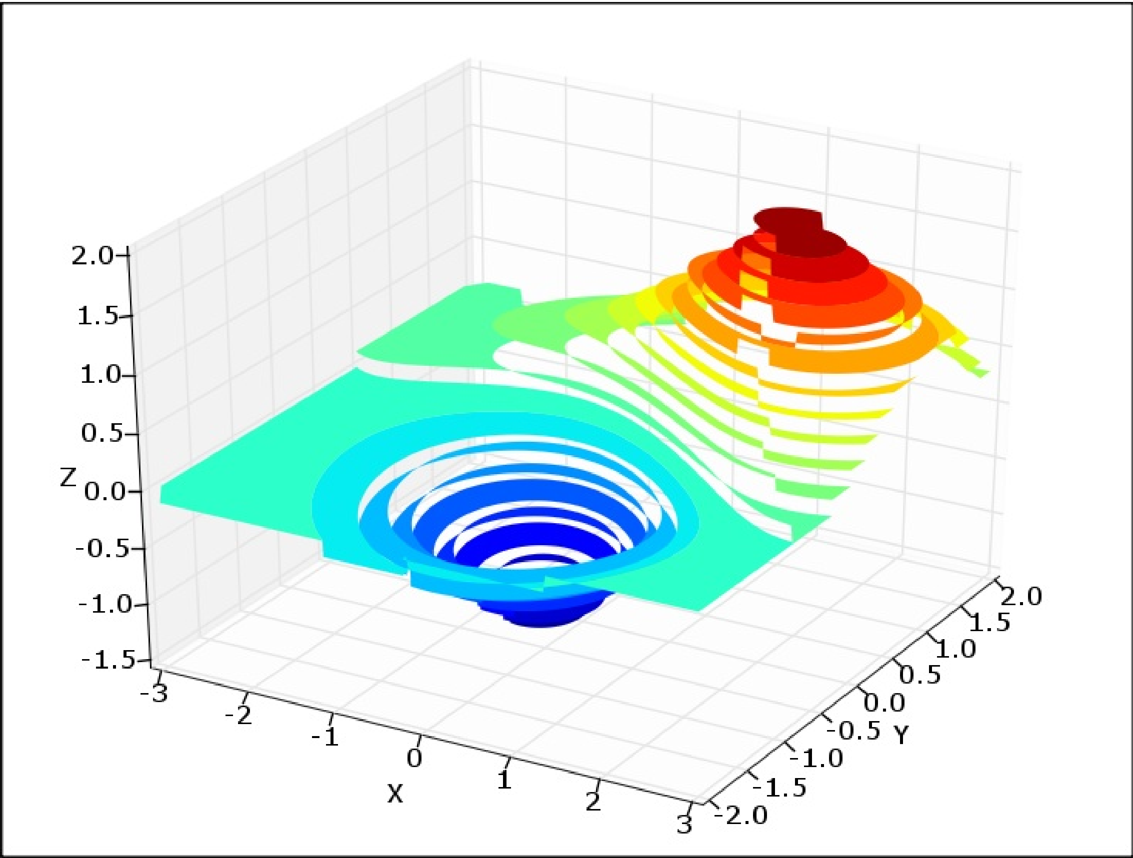


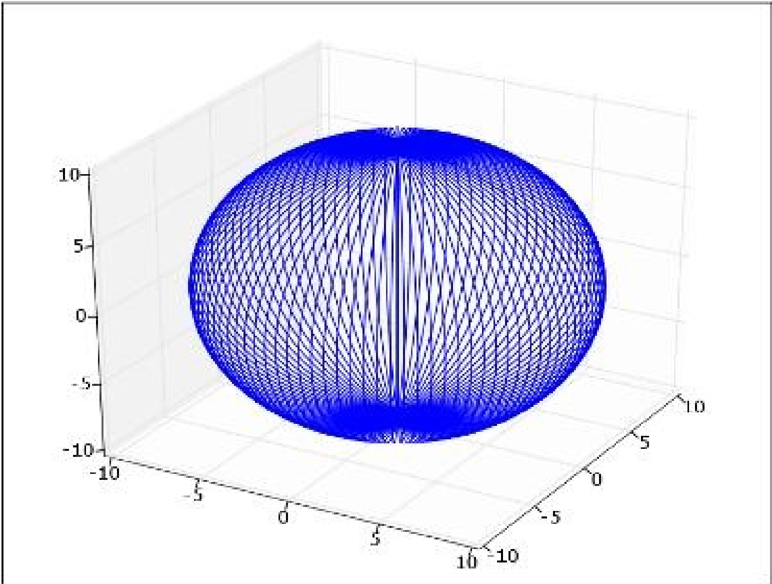

2.0
1.5
1.0
0.5
Z 0.0
-0.5
-1.0
-1.5
-3
-2
-1
0
1
2
3
X
2.0
1.5
1.0
0.5
0.0
-0.5 Y
-1.0
-1.5
-2.0

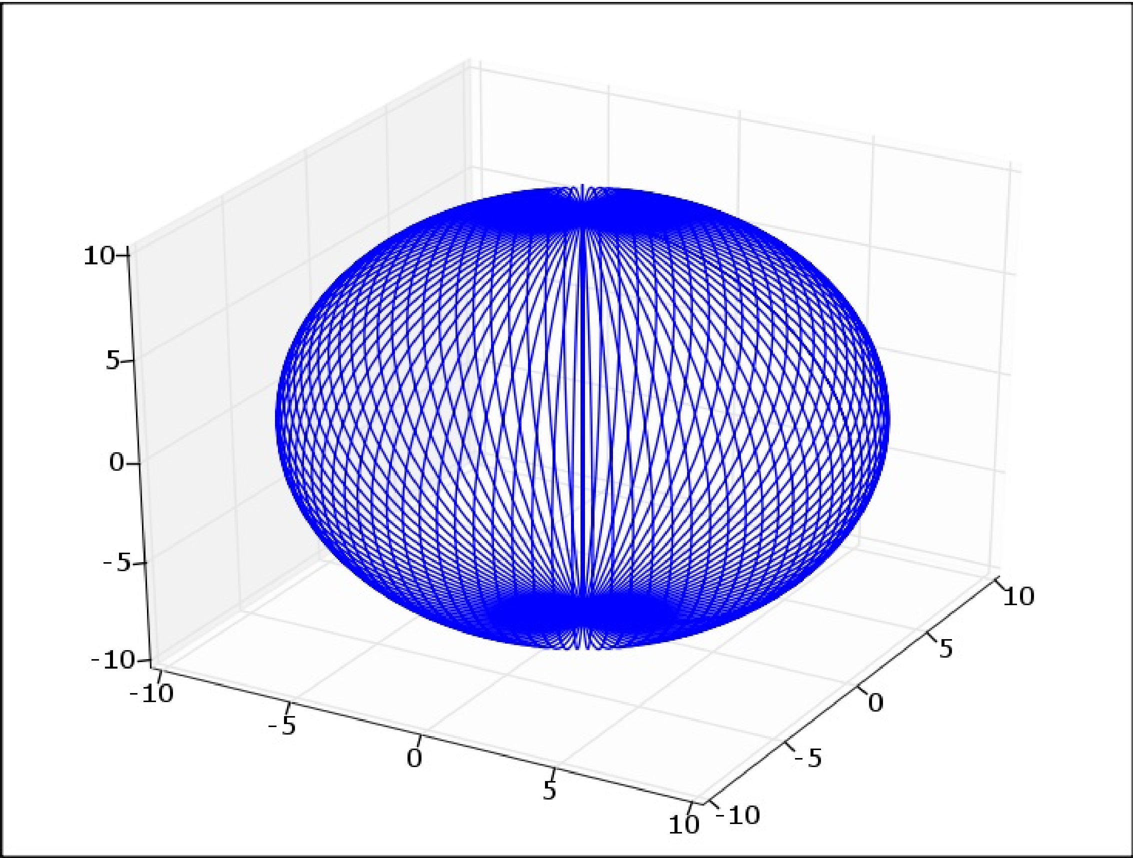


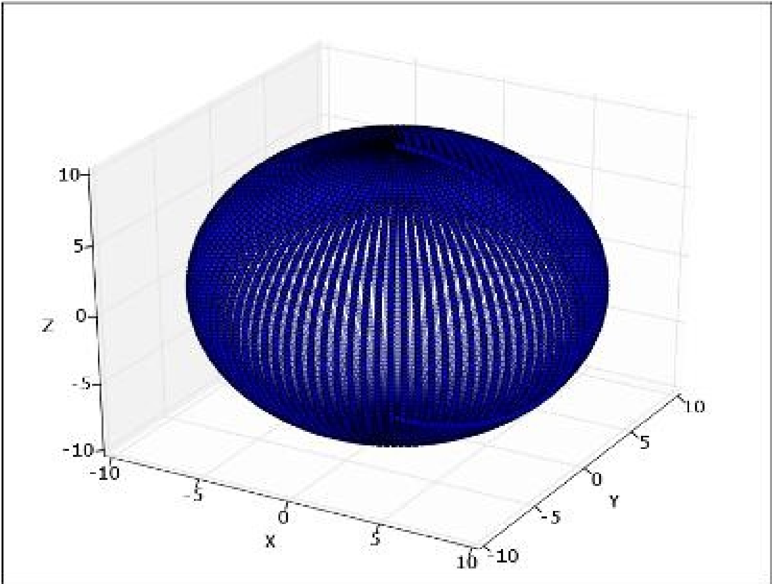

10
5
0
-5
-10
-10
-5
0
5
10

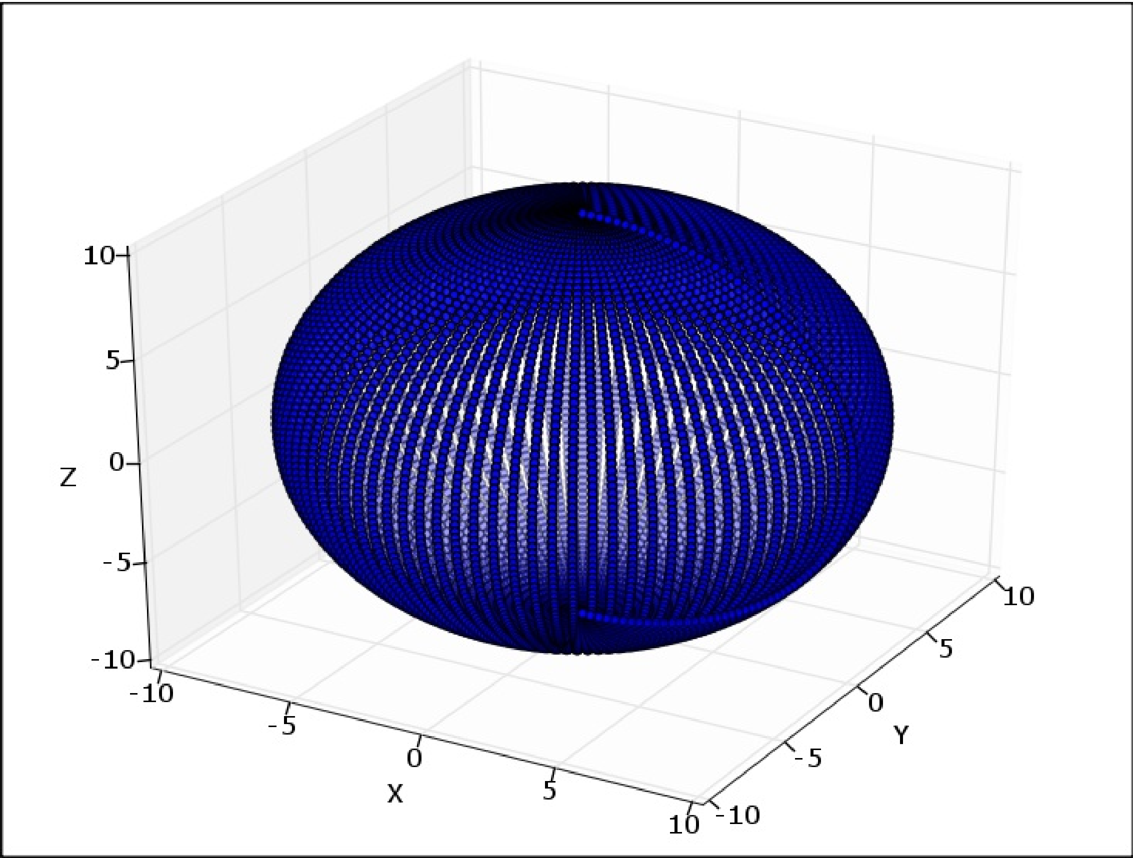

2.0
1.5
1.0
0.5
Z
0.0
-0.5
1.0
-1.5
-3
-2
1
0
1
2
3
X
2.0
1.5
1.0
0.5
0.0
-0.5
Y
-1.0
1.5
-2.0

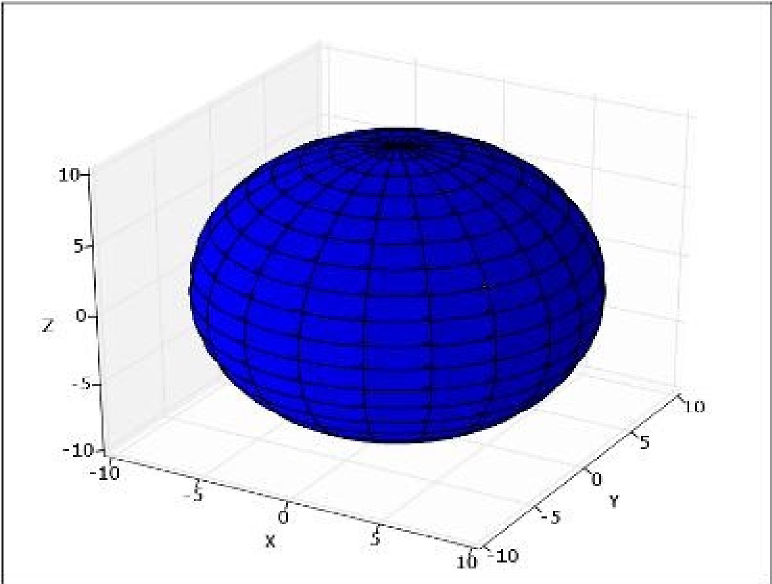

2.0
1.5
1.0
0.5
Z 0.0
-0.5
-1.0
-1.5
-3
-2
-1
0
1
2
3
X
2.0
1.5
1.0
0.5
0.0
-0.5 Y
-1.0
-1.5
-2.0

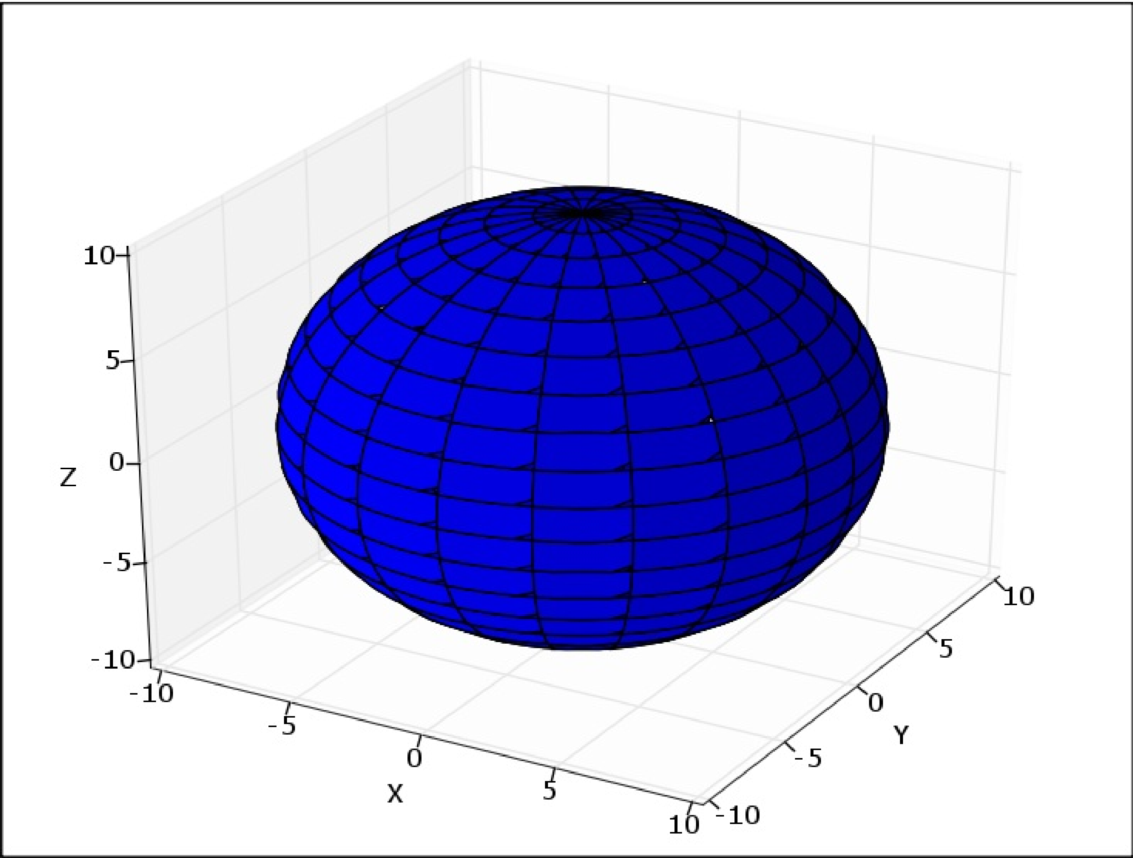

10
5
0
-5
-10
-10
-5
0
5
10
10
5
0
-5
-10

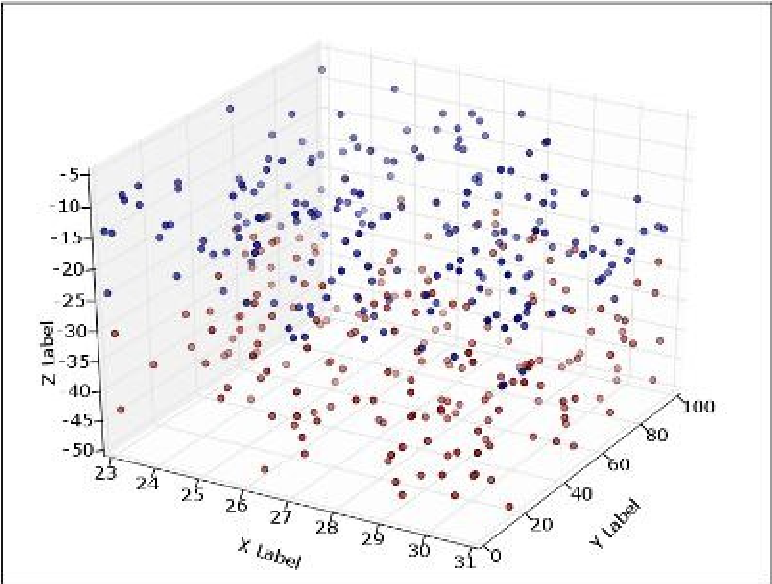

10
5
0
-5
-10
-10
-5
0
5
10
-10
-5
0
5
10

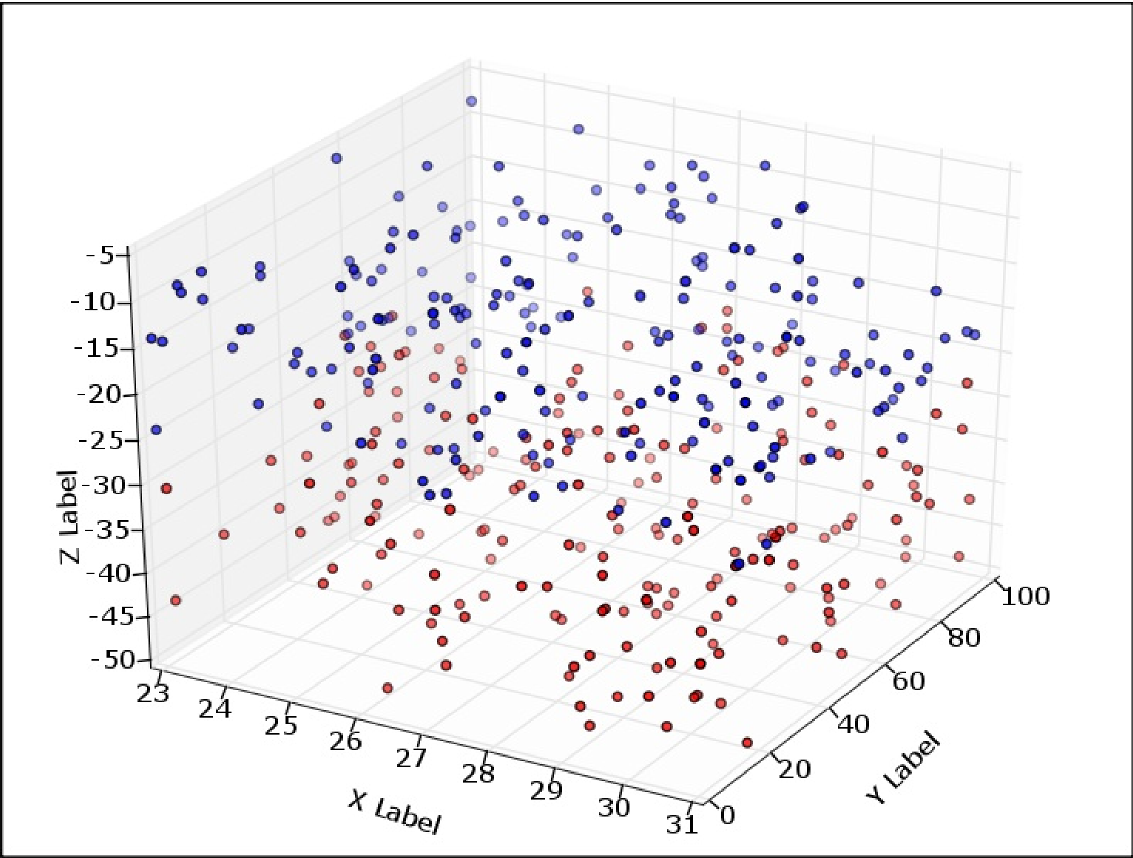

10
5
Z
0
-5
-10
-10
-5
0
X
5
10
-10
-5
0
Y
5
10

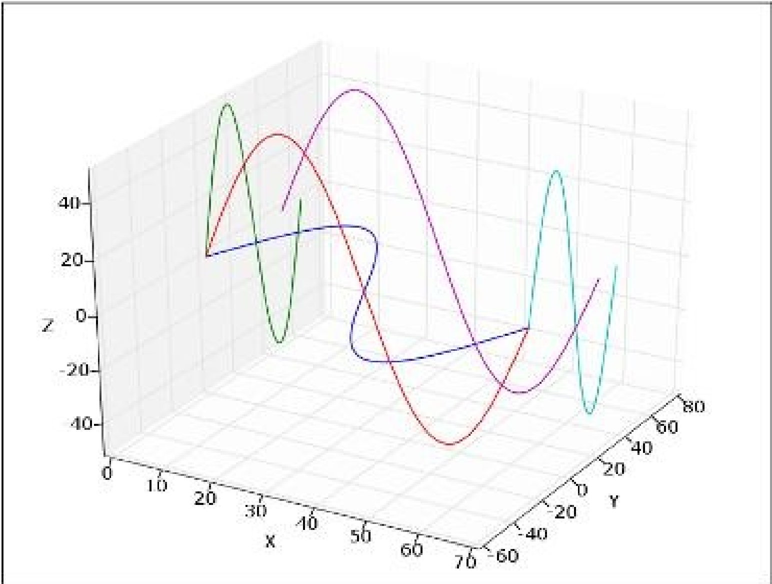

10
5
0
Z
-5
-10
-10
-5
0
X
5
10
-10
-5
0
Y
5
10

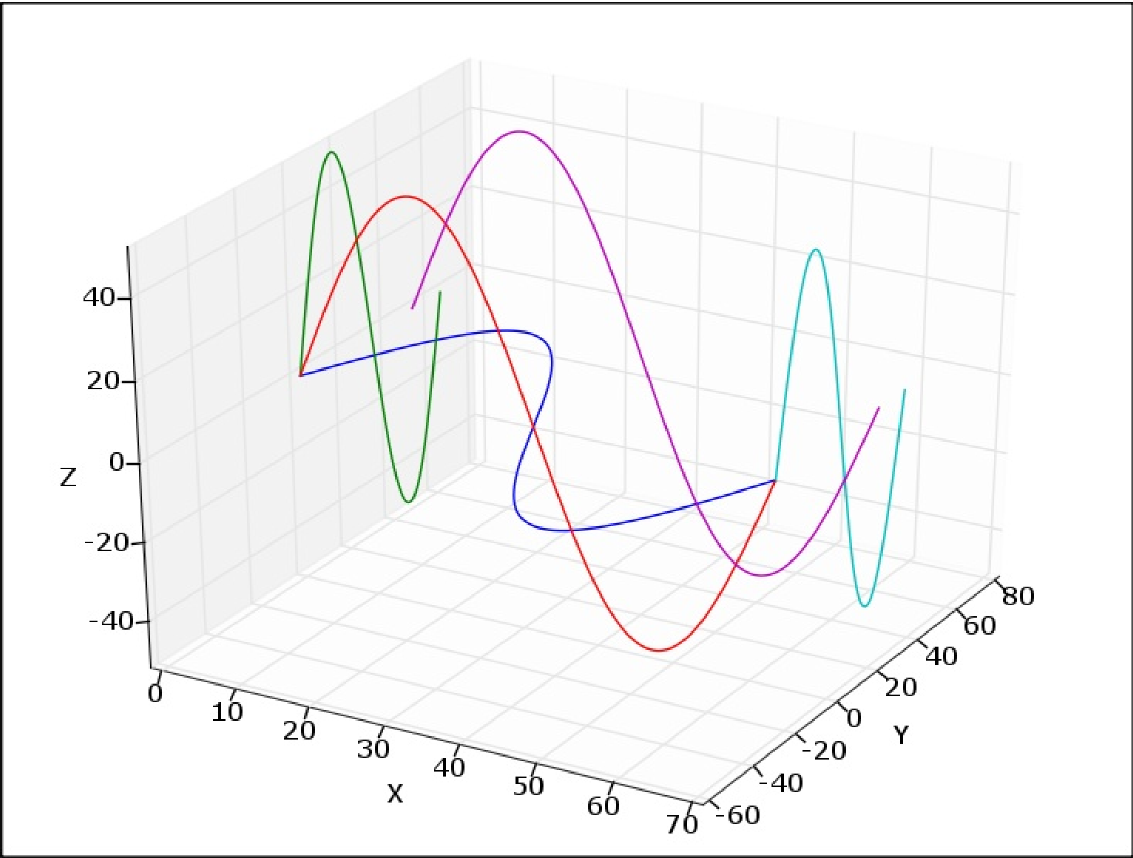

10
5
0
Z
-5
-10
-10
-5
0
X
5
10
-10
-5
0
Y
5
10

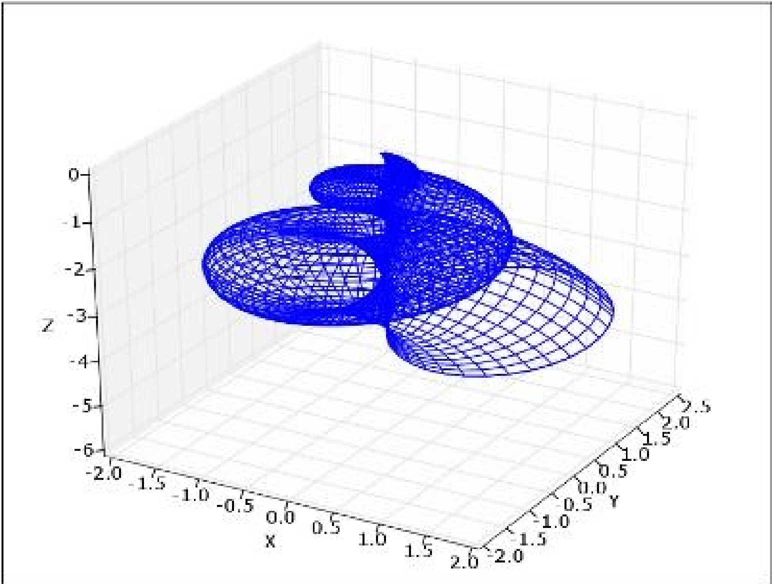

X
Y
Z

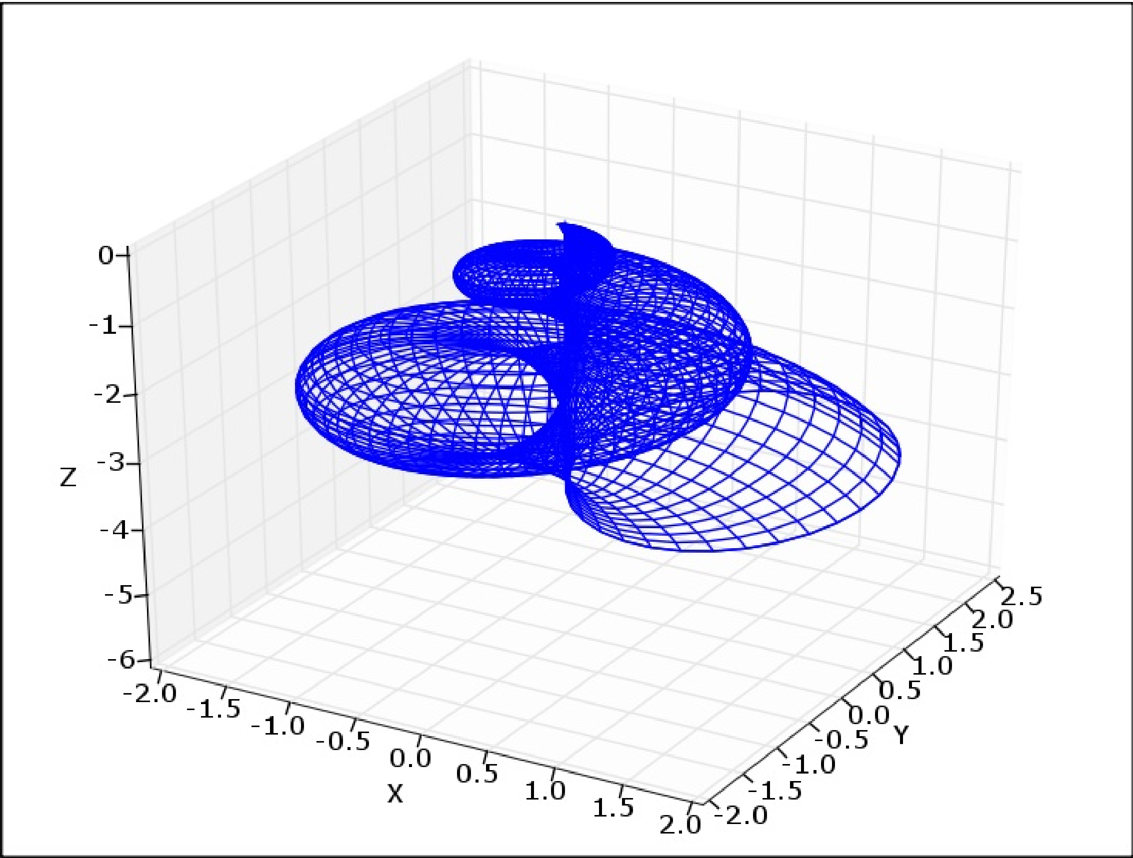

X Label
Y Label
Z Label

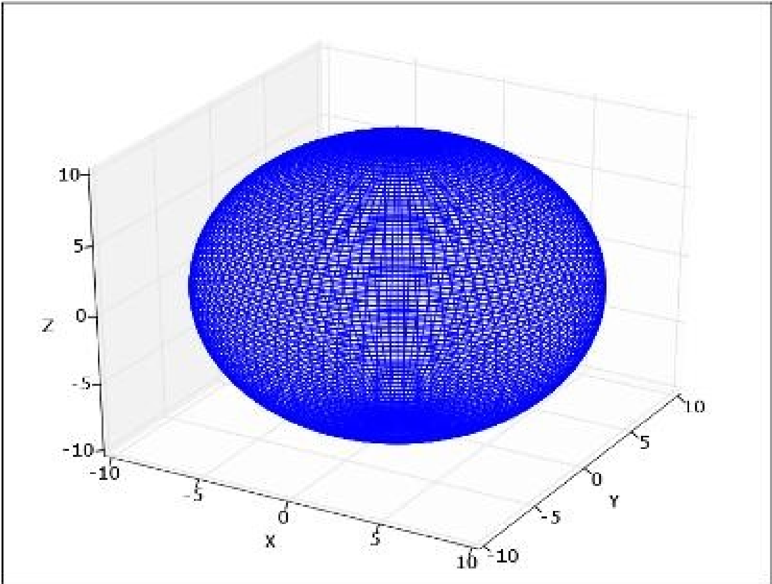

-5
-10
-15
-20
-25
-30
-35
-40
-45
-50
Z Label
23
24
25
26
27
28
29
30
31
X Label
0
20
40
60
80
100
Y Label

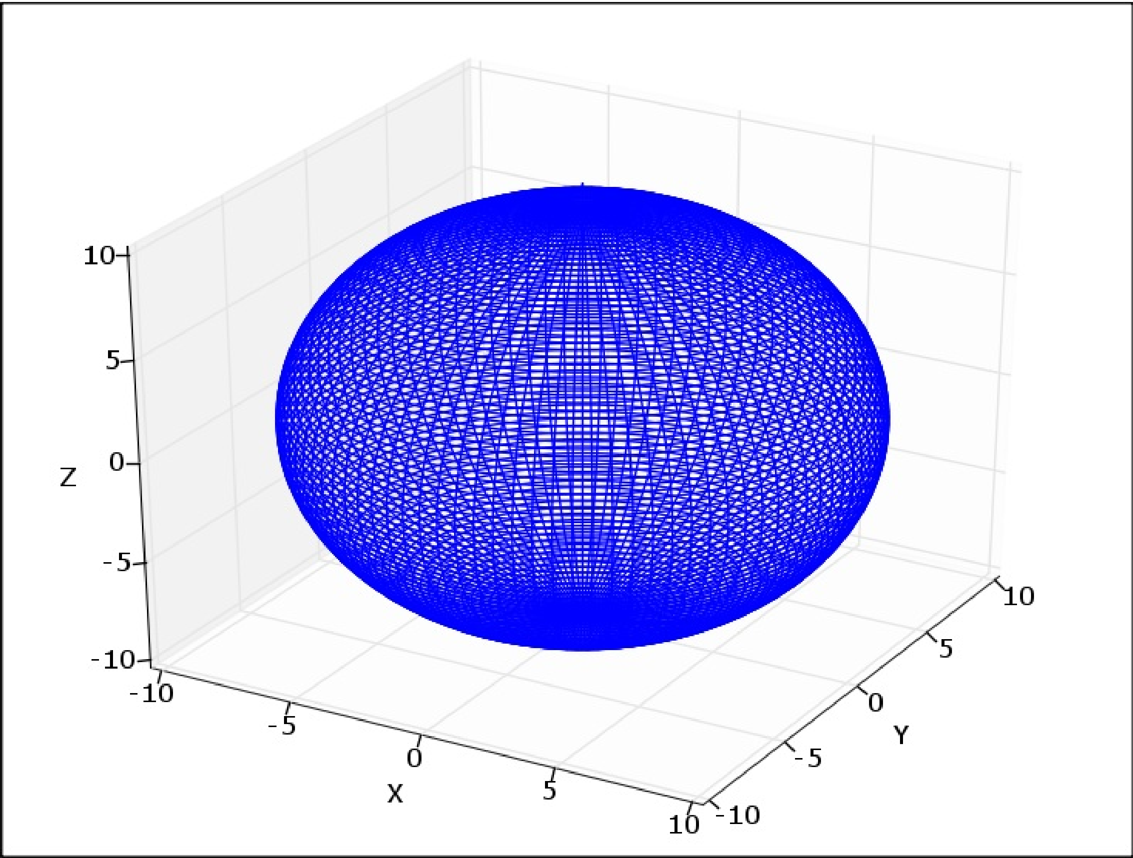

40
20
0
-20
-40
Z
0
10
20
30
40
50
60
70
X
-60
-40
-20
0
20
40
60
80
Y


40
20
0
-20
-40
Z
0
10
20
30
40
50
60
70
X
80
60
40
20
0
-20
-40
-60
Y




0
-1
-2
-3
-4
-5
-6
Z
-2.0
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5
2.0
X
2.5
2.0
1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
-2.0
Y


10
5
0
-5
-10
Z
-10
-5
0
5
10
X
10
5
0
-5
-10
Y


10
5
0
Z
-5
-10
-10
-5
0
X
5
10
-10
-5
0
Y
5
10

Matplotlib / Embedding Plots in Apps

Embedding In Wx

Matplotlib can be embedded in wxPython applications to provide high quality data visualization. There are two approaches to this, direct embedding and using an embedding library.

''Direct embedding'' is where your put one of the wxPython backend widgets (which subclass {{{wx.Panel}}}) directly into your application and draw plots on it using matplotlib's object-oriented API. This approach is demonstrated by the [http://cvs.sourceforge.net/viewcvs.py/matplotlib/matplotlib/examples/ embedding_in_wx*.py] examples that come with matplotlib. Neither {{{FigureCanvasWx}}} nor {{{FigureCanvasWxAgg}}} provide any facilities for user interactions like displaying the coordinates under the mouse, so you'll have to implement such things yourself. The matplotlib example [http://cvs.sourceforge.net/viewcvs.py/%2Acheckout%2A/matplotlib/matplotlib/examples/wxcursor_demo.py?content-type=text%2Fplain wxcursor_demo.py] should help you get started.

An ''embedding library'' saves you a lot of time and effort by providing plotting widgets that already support user interactions and other bells and whistles. There are two such libraries that I am aware of:

1. Matt Newville's [http://cars9.uchicago.edu/~newville/Python/MPlot/ MPlot] package supports drawing 2D line plots using pylab-style {{{plot()}}} and {{{oplot()}}} methods.
2. Ken !McIvor's [http://agni.phys.iit.edu/~kmcivor/wxmpl/ WxMpl] module supports drawing all plot types using matplotlib's object-oriented API.

Each of these libraries has different benefits and drawbacks, so I encourage you to evaluate each of them and select the one that best meets your needs.

Learning the Object-Oriented API

If you're embedding matplotlib in a wxPython program, you're probably going to have to use Matplotlib's Object-Oriented API to at some point. Take heart, as it matches the pylab API closely and is easy to pick up. There are more nuts and bolts to deal with, but that's no problem to someone already programming with wxPython! ;-)

The matplotlib FAQ [http://matplotlib.sourceforge.net/faq.html#OO" links] to several resources for learning about the OO API. Once you've got your feet wet, reading the classdocs is the most helpful source of information. The [http://matplotlib.sourceforge.net/matplotlib.axes.html#Axes" matplotlib.axes.Axes] class is where most of the plotting methods live, so it's a good place to start after you've conquored creating a Figure.

For your edification, a series of pylab examples have been translated to the OO API. They are available in a demonstration script that must be run from a command line. You may use any interactive matplotlib backend to display these plots.

A Simple Application

Here is a simple example of an application written in wx that embeds a [“Matplotlib figure in a wx panel”]. No toolbars, mouse clicks or any of that, just a plot drawn in a panel. Some work has been put into it to make sure that the figure is only redrawn once during a resize. For plots with many points, the redrawing can take some time, so it is best to only redraw when the figure is released. Have a read of the code.

Matplotlib: pyside

This is a very basic example showing how to display a matplotlib plot within a Qt application using PySide. In case of problems try to change the rcParam entry “backend.qt4” to “PySide” (e.g. by in the matplotlibrc file).

```
#!/usr/bin/env python
import sys
import matplotlib
matplotlib.use('Qt4Agg')
import pylab

from matplotlib.backends.backend_qt4agg import FigureCanvasQTAgg as
from matplotlib.figure import Figure

from PySide import QtCore, QtGui

if __name__ == '__main__':
    app = QtGui.QApplication(sys.argv)

    # generate the plot
    fig = Figure(figsize=(600,600), dpi=72, facecolor=(1,1,1), edge
    ax = fig.add_subplot(111)
    ax.plot([0,1])
    # generate the canvas to display the plot
    canvas = FigureCanvas(fig)

    win = QtGui.QMainWindow()
    # add the plot canvas to a window
    win.setCentralWidget(canvas)

    win.show()

    sys.exit(app.exec_())
```

Matplotlib: scrolling plot

Controlling an Embedded Plot with wx Scrollbars

When plotting a very long sequence in a matplotlib canvas embedded in a wxPython application, it sometimes is useful to be able to display a portion of the sequence without resorting to a scrollable window so that both axes remain visible. Here is how to do so:

```
from numpy import arange, sin, pi, float, size

import matplotlib
matplotlib.use('WXAgg')
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg
from matplotlib.figure import Figure

import wx

class MyFrame(wx.Frame):
    def __init__(self, parent, id):
        wx.Frame.__init__(self,parent, id, 'scrollable plot',
                style=wx.DEFAULT_FRAME_STYLE ^ wx.RESIZE_BORDER,
                size=(800, 400))
        self.panel = wx.Panel(self, -1)

        self.fig = Figure((5, 4), 75)
        self.canvas = FigureCanvasWxAgg(self.panel, -1, self.fig)
        self.scroll_range = 400
        self.canvas.SetScrollbar(wx.HORIZONTAL, 0, 5,
                                 self.scroll_range)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.canvas, -1, wx.EXPAND)

        self.panel.SetSizer(sizer)
        self.panel.Fit()

        self.init_data()
        self.init_plot()

        self.canvas.Bind(wx.EVT_SCROLLWIN, self.OnScrollEvt)

    def init_data(self):

        # Generate some data to plot:
        self.dt = 0.01
        self.t = arange(0,5,self.dt)
        self.x = sin(2*pi*self.t)
```

```
        # Extents of data sequence:
        self.i_min = 0
        self.i_max = len(self.t)

        # Size of plot window:
        self.i_window = 100

        # Indices of data interval to be plotted:
        self.i_start = 0
        self.i_end = self.i_start + self.i_window

    def init_plot(self):
        self.axes = self.fig.add_subplot(111)
        self.plot_data = \
                  self.axes.plot(self.t[self.i_start:self.i_end],
                                 self.x[self.i_start:self.i_end])[(

    def draw_plot(self):

        # Update data in plot:
        self.plot_data.set_xdata(self.t[self.i_start:self.i_end])
        self.plot_data.set_ydata(self.x[self.i_start:self.i_end])

        # Adjust plot limits:
        self.axes.set_xlim((min(self.t[self.i_start:self.i_end]),
                           max(self.t[self.i_start:self.i_end])))
        self.axes.set_ylim((min(self.x[self.i_start:self.i_end]),
                            max(self.x[self.i_start:self.i_end])))

        # Redraw:
        self.canvas.draw()

    def OnScrollEvt(self, event):

    # Update the indices of the plot:
        self.i_start = self.i_min + event.GetPosition()
        self.i_end = self.i_min + self.i_window + event.GetPositior
        self.draw_plot()

class MyApp(wx.App):
    def OnInit(self):
        self.frame = MyFrame(parent=None,id=-1)
        self.frame.Show()
        self.SetTopWindow(self.frame)
        return True

if __name__ == '__main__':
    app = MyApp()
    app.MainLoop()
```

Matplotlib / Misc

- Load image
- Matplotlib: adjusting image size
- Matplotlib: compiling matplotlib on solaris10
- Matplotlib: deleting an existing data series
- Matplotlib: django
- Matplotlib: interactive plotting
- Handling click events while zoomed
- Matplotlib: matplotlib and zope
- Matplotlib: multiple subplots with one axis label
- Matplotlib: qt with ipython and designer
- Matplotlib: using matplotlib in a CGI script

Load image

Image processing often works on gray scale images that were stored as PNG files. How do we import / export that file into {{{python}}}?

- Here is a recipy to do this with Matplotlib using the {{{imread}}} function (your image is called {{{lena.png}}}).

```
 from pylab import imread, imshow, gray, mean
a = imread('lena.png')
 # generates a RGB image, so do
aa=mean(a,2) # to get a 2-D array
imshow(aa)
gray()
```

This permits to do some processing for further exporting such as for [:Cookbook/Matplotlib/converting_a_matrix_to_a_raster_image:converting a matrix to a raster image]. In the newest version of pylab (check that your {{{pylab.matplotlib.**version**}}} is superior to {{{'0.98.0'}}}) you get directly a 2D numpy array if the image is grayscale.

- to write an image, do import Image mode = 'L' size= (256, 256) imNew=Image.new(mode , size) mat = numpy.random.uniform(size = size) data = numpy.ravel(mat) data = numpy.floor(data * 256)

imNew.putdata(data) imNew.save("rand.png")

- this kind of functions live also under {{{scipy.misc}}}, see for instance {{{scipy.misc.imsave}}} to create a color image: from scipy.misc import imsave import numpy a = numpy.zeros((4,4,3)) a[0,0,:] = [128, 0 , 255] imsave('graybackground_with_a_greyish_blue_square_on_top.png',a)
- to define the range, use: from scipy.misc import toimage import numpy a = numpy.random.rand(25,50) #between 0. and 1. toimage(a, cmin=0., cmax=2.).save('low_contrast_snow.png') (adapted from http://telin.ugent.be/~slippens/drupal/scipy_unscaledimsave)
- there was another (more direct) method suggested by http://jehiah.cz/archive/creating-images-with-numpy

Matplotlib: adjusting image size

This is a small demo file that helps teach how to adjust figure sizes for matplotlib

First a little introduction

There are three parameters define an image size (this is not MPL specific):

```
* Size in length units (inches, cm, pt, etc): e.g. 5"x7"``* Size in
```

```
Only two of these are independent, so if you define two of them, the
```

When displaying on a computer screen (or saved to a PNG), the size in length units is irrelevant, the pixels are simply displayed. When printed, or saved to PS, EPS or PDF (all designed to support printing), then the Size or dpi is used to determine how to scale the image.

Now I'm getting into how MPL works

```
. 1) The size of a figure is defined in length units (inches), and
```

```
The trick here is that when printing, it's natural to think in terms
```

Another trick

Figure.savefig() overrides the dpi setting in figure, and uses a default (which on my system at least is 100 dpi). If you want to overide it, you can specify the 'dpi' in the savefig call:

The following code will hopefully make this more clear, at least for generating PNGs for web pages and the like.

`MPL_size_test.py`

```
#!python

"""
This is a small demo file that helps teach how to adjust figure siz
for matplotlib

"""

import matplotlib
print "using MPL version:", matplotlib.__version__
matplotlib.use("WXAgg") # do this before pylab so you don'tget the

import pylab
import matplotlib.numerix as N

# Generate and plot some simple data:
x = N.arange(0, 2*N.pi, 0.1)
y = N.sin(x)

pylab.plot(x,y)
F = pylab.gcf()

# Now check everything with the defaults:
DPI = F.get_dpi()
print "DPI:", DPI
DefaultSize = F.get_size_inches()
print "Default size in Inches", DefaultSize
print "Which should result in a %i x %i Image"%(DPI*DefaultSize[0],
# the default is 100dpi for savefig:
F.savefig("test1.png")
# this gives me a 797 x 566 pixel image, which is about 100 DPI

# Now make the image twice as big, while keeping the fonts and all
# same size
F.set_figsize_inches( (DefaultSize[0]*2, DefaultSize[1]*2) )
Size = F.get_size_inches()
print "Size in Inches", Size
F.savefig("test2.png")
# this results in a 1595x1132 image

# Now make the image twice as big, making all the fonts and lines
# bigger too.

F.set_figsize_inches( DefaultSize )# resetthe size
Size = F.get_size_inches()
print "Size in Inches", Size
F.savefig("test3.png", dpi = (200)) # change the dpi
# this also results in a 1595x1132 image, but the fonts are larger
```

Putting more than one image in a figure

Suppose you have two images: 100x100 and 100x50 that you want to display in a figure with a buffer of 20 pixels (relative to image pixels) between them and a border of 10 pixels all around.

The solution isn't particularly object oriented, but at least it gets to the practical details.

```
#!python
def _calcsize(matrix1, matrix2, top=10, left=10, right=10, bottom=1
    size1 = array(matrix1.shape) * scale
    size2 = array(matrix2.shape) * scale
    _width = float(size1[1] + size2[1] + left + right + buffer)
    _height = float(max(size1[0], size2[0]) + top + bottom)
    x1 = left / _width
    y1 = bottom / _height
    dx1 = size1[1] / _width
    dy1 = size1[0] / _height
    size1 = (x1, y1, dx1, dy1)
    x2 = (size1[1] + left + buffer) / _width
    y2 = bottom / _height
    dx2 = size2[1] / _width
    dy2 = size2[0] / _height
    size2 = (x2, y2, dx2, dy2)
    figure = pylab.figure(figsize=(_width * height / _height, height
    axis1 = apply(pylab.axes, size1)
    pylab.imshow(X1, aspect='preserve')
    axis2 = apply(pylab.axes, size2)
    pylab.imshow(X2, aspect='preserve')
    return axes1, axes2, figure
```

Attachments

- `MPL_size_test.py`

Matplotlib: compiling matplotlib on solaris10

how to install sunstudio and build matplotlib on solaris 10 might give some hints.

JDH said:

> Hi Erik – if you succeed, then we'll have convincing proof that compiling mpl on solaris is easier than giving up the sauce.

Well, it has turned out to be easier than giving up the sauce (at least for me), but only by a whisker. In the end, the fix is incredibly simple (if you consider recompiling python and manually adjusting the auto-produced pyconfig.h incredibly simple, anyway). After two solid days of commenting this and that out, recompiling everything and its mother 76 different ways from Sunday, poring over a legion of Solaris sys includes, slaughtering a few spotlessly white lambs and one pure black sheep, wrapping the bones and tendons and viscera in a double layer of fat and burning the offering to Delphic Apollo, I found the answer:

1 download Python 2.4.2

2 after extracting it and running ./configure, edit the generated pyconfig.h as follows:

```
i) if _XOPEN_SOURCE is defined to be 600 (i.e., if the line "#defi

ii) if _XOPEN_SOURCE_EXTENDED is defined at all (i.e. if the line '
```

3 make && make install

The problem was with Solaris's support for the X/Open standards. To make a long story short, you can use Open Group Technical Standard, Issue 6 (XPG6/UNIX 03/SUSv3) (_XOPEN_SOURCE == 600) if and only if you are using an ISO C99 compiler. If you use X/Open CAE Specification, Issue 5 (XPG5/UNIX 98/SUSv2) (_XOPEN_SOURCE == 500), you don't have to use an ISO C99 compiler. For full details, see the Solaris header file /usr/include/sys/feature_tests.h.

This is why muhpubuh (AKA matplotlib—long story) compiles on Solaris 10 if you have the big bucks and can afford Sun's OpenStudio 10 compiler. gcc does not have full C99 support yet. In particular, its lack of support for the wide character library makes the build go bust. (See, e.g., http://gcc.gnu.org/c99status.html.)

More helpful links on the wchar problem with Python.h and Solaris :

- http://lists.schmorp.de/pipermail/rxvt-unicode/2005q2/000092.html
- http://bugs.opensolaris.org/bugdatabase/view_bug.do?bug_id=6395191
- http://mail.python.org/pipermail/patches/2005-June/017820.html
- http://mail.python.org/pipermail/python-bugs-list/2005-November/030900.html

Matplotlib: deleting an existing data series

Each axes instance contains a lines attribute, which is a list of the data series in the plot, added in chronological order. To delete a particular data series, one must simply delete the appropriate element of the lines list and redraw if necessary.

The is illustrated in the following example from an interactive session:

```
>>> x = N.arange(10)

>>> fig = P.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(x)
[<matplotlib.lines.Line2D instance at 0x427ce7ec>]

>>> ax.plot(x+10)
[<matplotlib.lines.Line2D instance at 0x427ce88c>]

>>> ax.plot(x+20)
[<matplotlib.lines.Line2D instance at 0x427ce9ac>]

>>> P.show()
>>> ax.lines
[<matplotlib.lines.Line2D instance at 0x427ce7ec>,
 <matplotlib.lines.Line2D instance at 0x427ce88c>,
 <matplotlib.lines.Line2D instance at 0x427ce9ac>]

>>> del ax.lines[1]
>>> P.show()
```

which will plot three lines, and then delete the second.

Matplotlib: django

1.
    i. Please edit system and help pages ONLY in the moinmaster wiki! For more
    ii. information, please see MoinMaster:MoinPagesEditorGroup.
    iii. master-page:Unknown-Page
    iv. master-date:Unknown-Date
    v. acl MoinPagesEditorGroup:read,write,delete,revert All:read

Using MatPlotLib to dynamically generate charts in a Django web service

You need to have a working Django installation, plus matplotlib.

Example 1 - PIL Buffer

```
# file charts.py
def simple(request):
    import random
    import django
    import datetime

    from matplotlib.backends.backend_agg import FigureCanvasAgg as
    from matplotlib.figure import Figure
    from matplotlib.dates import DateFormatter

    fig=Figure()
    ax=fig.add_subplot(111)
    x=[]
    y=[]
    now=datetime.datetime.now()
    delta=datetime.timedelta(days=1)
    for i in range(10):
        x.append(now)
        now+=delta
        y.append(random.randint(0, 1000))
    ax.plot_date(x, y, '-')
    ax.xaxis.set_major_formatter(DateFormatter('%Y-%m-%d'))
    fig.autofmt_xdate()
    canvas=FigureCanvas(fig)
    response=django.http.HttpResponse(content_type='image/png')
    canvas.print_png(response)
    return response
```

Since some versions of Internet Explorer ignore the content_type. The URL should end with ".png". You can create an entry in your urls.py like this:

```
...
(r'^charts/simple.png$', 'myapp.views.charts.simple'),
...
```

Matplotlib: interactive plotting

Interactive point identification

I find it often quite useful to be able to identify points within a plot simply by clicking. This recipe provides a fairly simple functor that can be connected to any plot. I've used it with both scatter and standard plots.

Because often you'll have multiple views of a dataset spread across either multiple figures, or at least multiple axis, I've also provided a utility to link these plots together so clicking on a point in one plot will highlight and identify that data point on all other linked plots.

```
import math

import matplotlib.pyplot as plt

class AnnoteFinder(object):
    """callback for matplotlib to display an annotation when points
 clicked on.  The point which is closest to the click and within
 xtol and ytol is identified.

 Register this function like this:

 scatter(xdata, ydata)
 af = AnnoteFinder(xdata, ydata, annotes)
 connect('button_press_event', af)
 """

    def __init__(self, xdata, ydata, annotes, ax=None, xtol=None, y
        self.data = list(zip(xdata, ydata, annotes))
        if xtol is None:
            xtol = ((max(xdata) - min(xdata))/float(len(xdata)))/2
        if ytol is None:
            ytol = ((max(ydata) - min(ydata))/float(len(ydata)))/2
        self.xtol = xtol
        self.ytol = ytol
        if ax is None:
            self.ax = plt.gca()
        else:
            self.ax = ax
        self.drawnAnnotations = {}
        self.links = []

    def distance(self, x1, x2, y1, y2):
        """
 return the distance between two points
 """
```

```
        return(math.sqrt((x1 - x2)**2 + (y1 - y2)**2))

    def __call__(self, event):

        if event.inaxes:

            clickX = event.xdata
            clickY = event.ydata
            if (self.ax is None) or (self.ax is event.inaxes):
                annotes = []
                # print(event.xdata, event.ydata)
                for x, y, a in self.data:
                    # print(x, y, a)
                    if ((clickX-self.xtol < x < clickX+self.xtol) a
                            (clickY-self.ytol < y < clickY+self.ytc
                        annotes.append(
                            (self.distance(x, clickX, y, clickY), x
                if annotes:
                    annotes.sort()
                    distance, x, y, annote = annotes[0]
                    self.drawAnnote(event.inaxes, x, y, annote)
                    for l in self.links:
                        l.drawSpecificAnnote(annote)

    def drawAnnote(self, ax, x, y, annote):
        """
  Draw the annotation on the plot
  """
        if (x, y) in self.drawnAnnotations:
            markers = self.drawnAnnotations[(x, y)]
            for m in markers:
                m.set_visible(not m.get_visible())
            self.ax.figure.canvas.draw_idle()
        else:
            t = ax.text(x, y, " - %s" % (annote),)
            m = ax.scatter([x], [y], marker='d', c='r', zorder=100)
            self.drawnAnnotations[(x, y)] = (t, m)
            self.ax.figure.canvas.draw_idle()

    def drawSpecificAnnote(self, annote):
        annotesToDraw = [(x, y, a) for x, y, a in self.data if a ==
        for x, y, a in annotesToDraw:
            self.drawAnnote(self.ax, x, y, a)
```

To use this functor you can simply do something like this:

```
x = range(10)
y = range(10)
annotes = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

fig, ax = plt.subplots()
ax.scatter(x,y)
af =  AnnoteFinder(x,y, annotes, ax=ax)
fig.canvas.mpl_connect('button_press_event', af)
plt.show()
```

This is fairly useful, but sometimes you'll have multiple views of a dataset and it is useful to click and identify a point in one plot and find it in another. The below code demonstrates this linkage and should work between multiple axis or figures.

```
def linkAnnotationFinders(afs):
  for i in range(len(afs)):
    allButSelfAfs = afs[:i]+afs[i+1:]
    afs[i].links.extend(allButSelfAfs)

subplot(121)
scatter(x,y)
af1 = AnnoteFinder(x,y, annotes)
connect('button_press_event', af1)

subplot(122)
scatter(x,y)
af2 = AnnoteFinder(x,y, annotes)
connect('button_press_event', af2)

linkAnnotationFinders([af1, af2])
```

I find this fairly useful. By subclassing and redefining drawAnnote this simple framework could be used to drive a more sophisticated user interface.

Currently this implementation is a little slow when the number of datapoints becomes large. I'm particularly interested in suggestions people might have for making this faster and better.

Matplotlib: matplotlib and zope

```
0. Prerequisites: You need to have the following installed to succe
```

```
1. Create a file (e.g. mpl.py) in INSTANCEHOME\Extensions:
```

```
import matplotlib
matplotlib.use('Agg')
from pylab import *
from os import *
from StringIO import StringIO
from PIL import Image as PILImage
from matplotlib.backends.backend_agg import FigureCanvasAgg
def chart(self):
    clf()
    img_dpi=72
    width=400
    height=300
    fig=figure(dpi=img_dpi, figsize=(width/img_dpi, height/img_dpi
    x=arange(0, 2*pi+0.1, 0.1)
    sine=plot(x, sin(x))
    legend(sine, "y=sin x", "upper right")
    xlabel('x')
    ylabel('y=sin x')
    grid(True)
    canvas = FigureCanvasAgg(fig)
    canvas.draw()
    size = (int(canvas.figure.get_figwidth())*img_dpi, int(canvas.f
    buf=canvas.tostring_rgb()
    im=PILImage.fromstring('RGB', size, buf, 'raw', 'RGB', 0, 1)
    imgdata=StringIO()
    im.save(imgdata, 'PNG')
    self.REQUEST.RESPONSE.setHeader('Pragma', 'no-cache')
    self.REQUEST.RESPONSE.setHeader('Content-Type', 'image/png')
    return imgdata.getvalue()
```

1. Then create an external method in ZMI (e.g. Id -> mplchart, module name -> mpl, function name -> chart).
2. Click the Test tab and you should see the sine plot.

Matplotlib: multiple subplots with one axis label

Using a single axis label to annotate multiple subplot axes

When using multiple subplots with the same axis units, it is redundant to label each axis individually, and makes the graph overly complex. You can use a single axis label, centered in the plot frame, to label multiple subplot axes. Here is how to do it:

```
#!python
# note that this a code fragment...you will have to define your own
# Set up a whole-figure axes, with invisible axis, ticks, and tickl
# which we use to get the xlabel and ylabel in the right place
bigAxes = pylab.axes(frameon=False)     # hide frame
pylab.xticks([])                        # don't want to see any tic
pylab.yticks([])
# I'm using TeX for typesetting the labels--not necessary
pylab.ylabel(r'\textbf{Surface Concentration $(nmol/m^2)$}', size=
pylab.xlabel(r'\textbf{Time (hours)}', size='medium')
# Create subplots and shift them up and to the right to keep tick l
# from overlapping the axis labels defined above
topSubplot = pylab.subplot(2,1,1)
position = topSubplot.get_position()
position[0] = 0.15
position[1] = position[1] + 0.01
topSubplot.set_position(position)
pylab.errorbar(times150, average150)
bottomSubplot = pylab.subplot(2,1,2)
position = bottomSubplot.get_position()
position[0] = 0.15
position[1] = position[1] + 0.03
bottomSubplot.set_position(position)
pylab.errorbar(times300, average300)
```

Alternatively, you can use the following snippet to have shared ylabels on your subplots. Also see the attached figure output.)#

```
#!python
import pylab

figprops = dict(figsize=(8., 8\. / 1.618), dpi=128)
adjustprops = dict(left=0.1, bottom=0.1, right=0.97, top=0.93, wspa

fig = pylab.figure(**figprops)
fig.subplots_adjust(**adjustprops)

ax = fig.add_subplot(3, 1, 1)
bx = fig.add_subplot(3, 1, 2, sharex=ax, sharey=ax)
cx = fig.add_subplot(3, 1, 3, sharex=ax, sharey=ax)

ax.plot([0,1,2], [2,3,4], 'k-')
bx.plot([0,1,2], [2,3,4], 'k-')
cx.plot([0,1,2], [2,3,4], 'k-')

pylab.setp(ax.get_xticklabels(), visible=False)
pylab.setp(bx.get_xticklabels(), visible=False)

bx.set_ylabel('This is a long label shared among more axes', fontsi
cx.set_xlabel('And a shared x label', fontsize=14)
```

Thanks to Sebastian Krieger from matplotlib-users list for this trick.

Simple function to get rid of superfluous xticks but retain the ones on the bottom (works in pylab). Combine it with the above snippets to get a nice plot without too much redundance:

```
#!python
def rem_x():
    '''Removes superfluous x ticks when multiple subplots  share
    their axis works only in pylab mode but can easily be rewritten
    for api use'''
    nr_ax=len(gcf().get_axes())
    count=0
    for z in gcf().get_axes():
        if count == nr_ax-1: break
            setp(z.get_xticklabels(),visible=False)
            count+=1
```

The first one above doesn’t work for me. The subplot command overwrites the bigaxes. However, I found a much simpler solution to do a decent job for two axes and one ylabel:

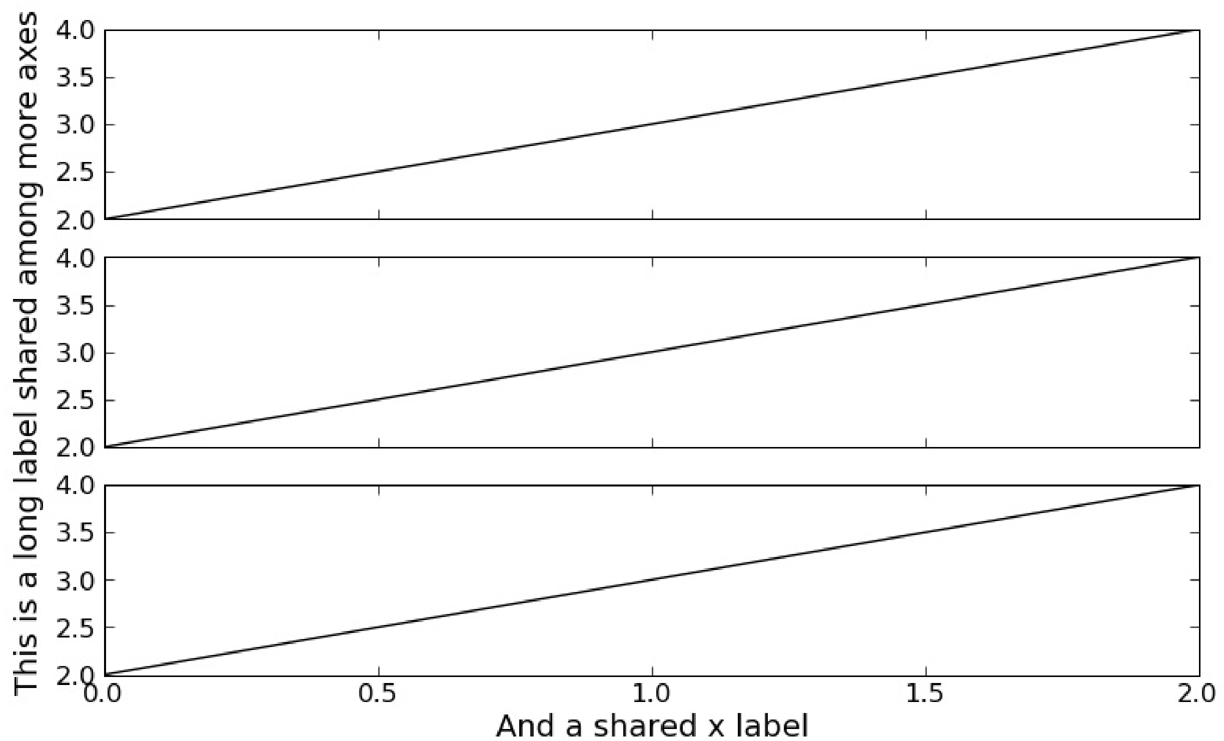
yyl=plt.ylabel(r’My longish label that I want vertically centred’)

yyl.set_position((yyl.get_position()[0],1)) # This says use the top of the bottom axis as the reference point.

yyl.set_verticalalignment('center')

Attachments

- `Same_ylabel_subplots.png`



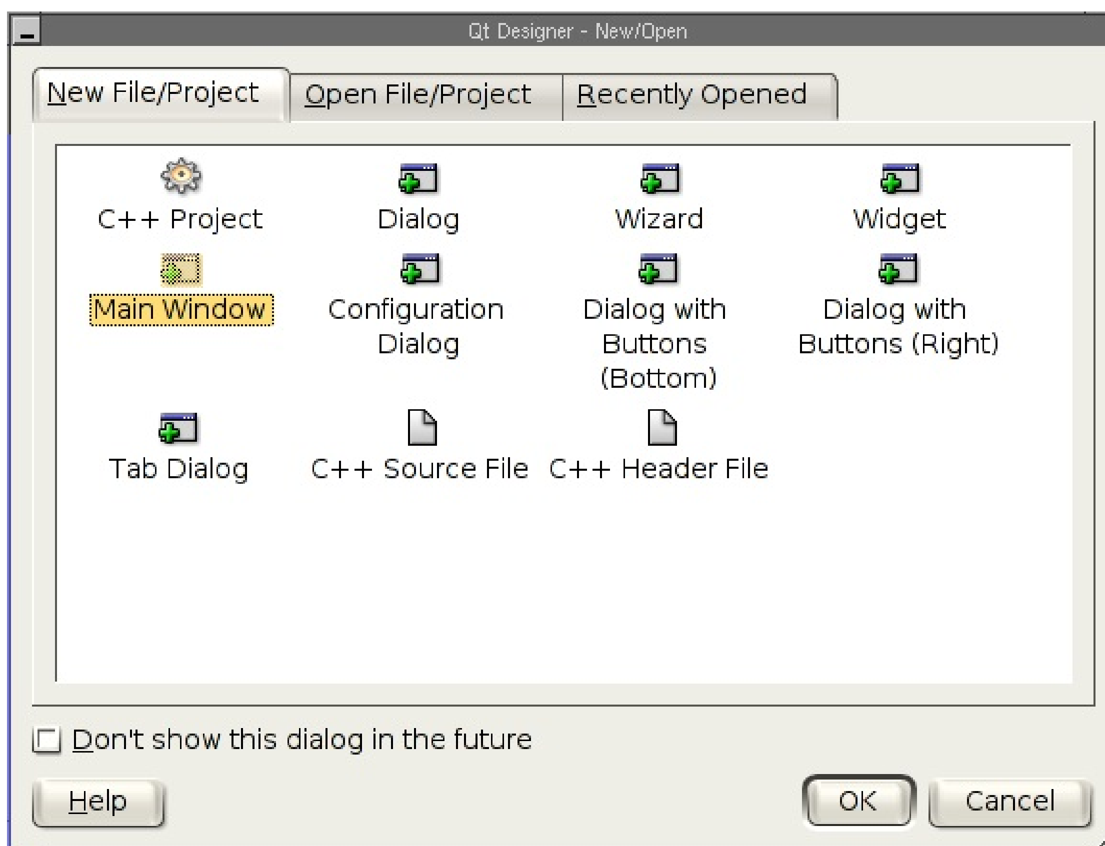
Matplotlib: qt with ipython and designer

Example code for embedding Matplotlib in Qt applications is given at embedding_in_qt.py. This recipe extends that basic formula for integration with other powerful tools. In particular, we bring together the use of the GUI creation tool made by Trolltech (the creators of Qt) and the ability to interact with a running Qt application via IPython.

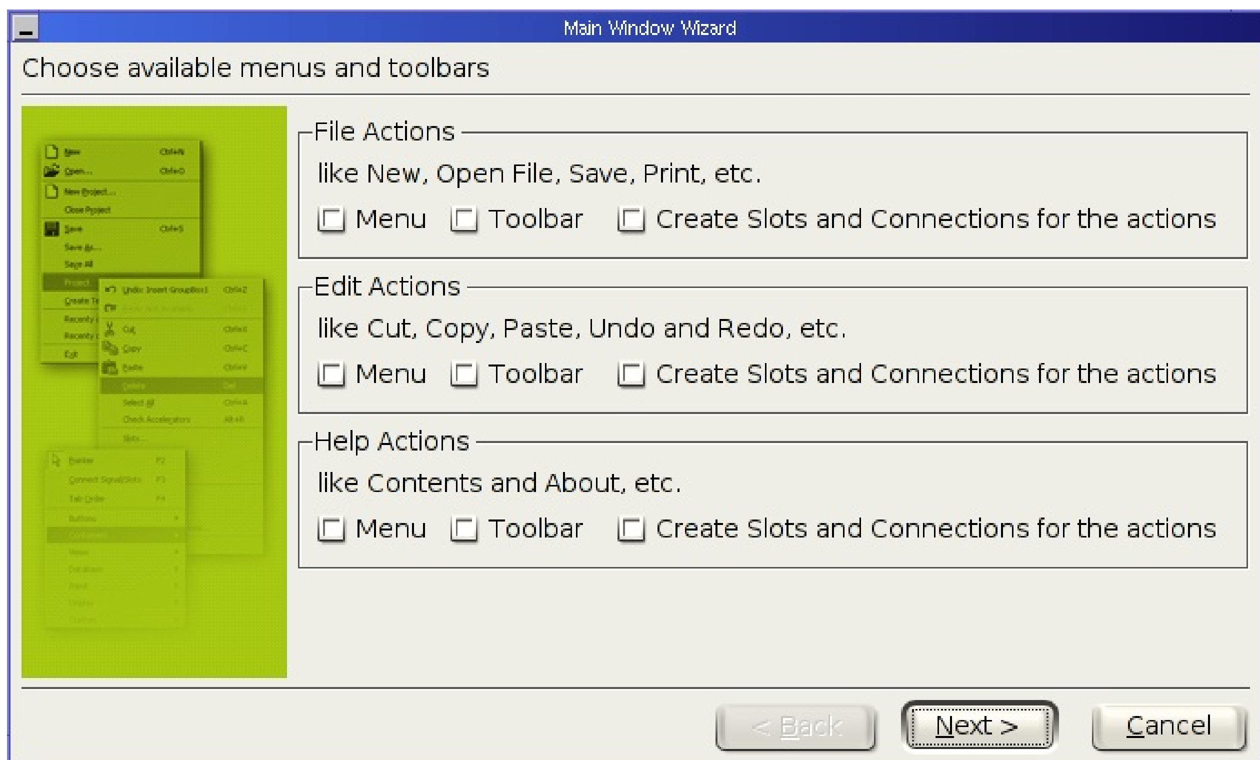
A basic tutorial for using Designer (along with the pertinent system requirements) has been posted by Alex Fedosov. Please review it before continuing.

However, for our purposes, we will create a much simpler design in Designer.

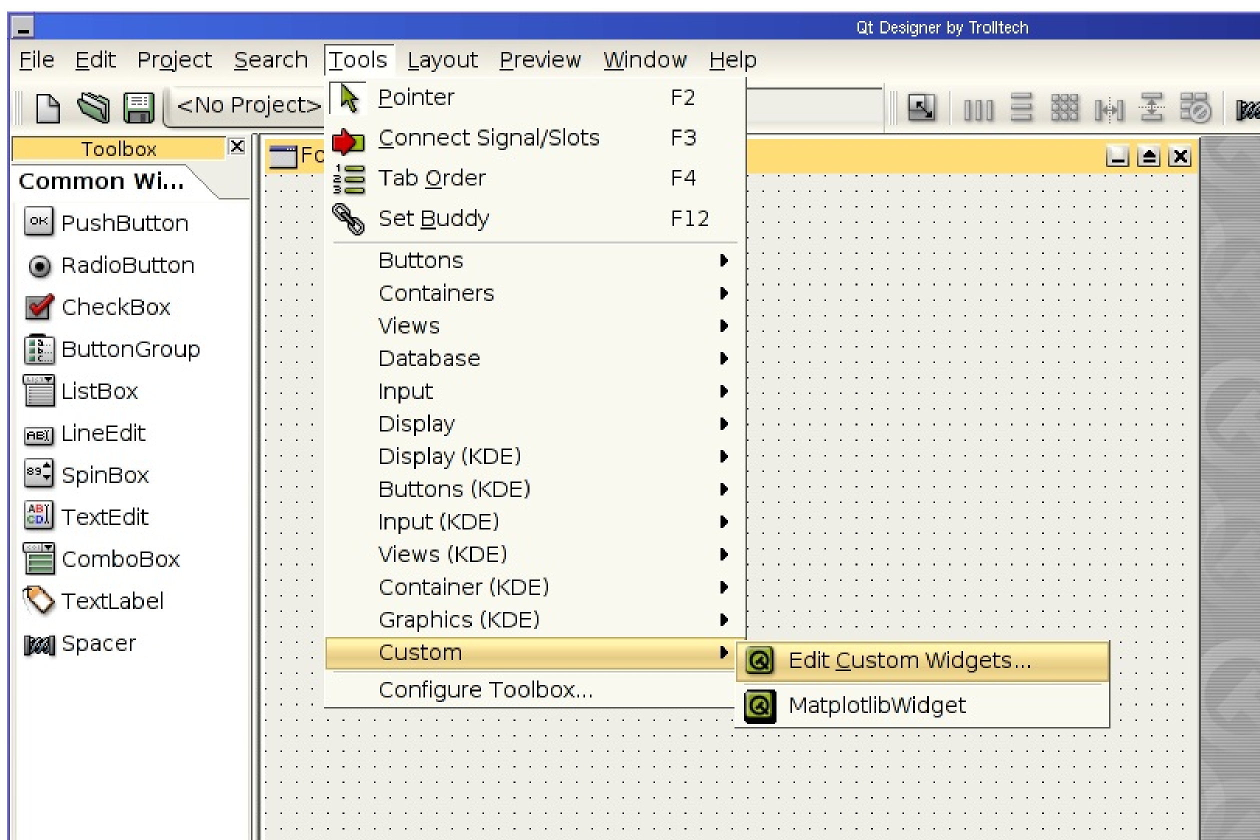
Open Designer, and create a new "Main Window":



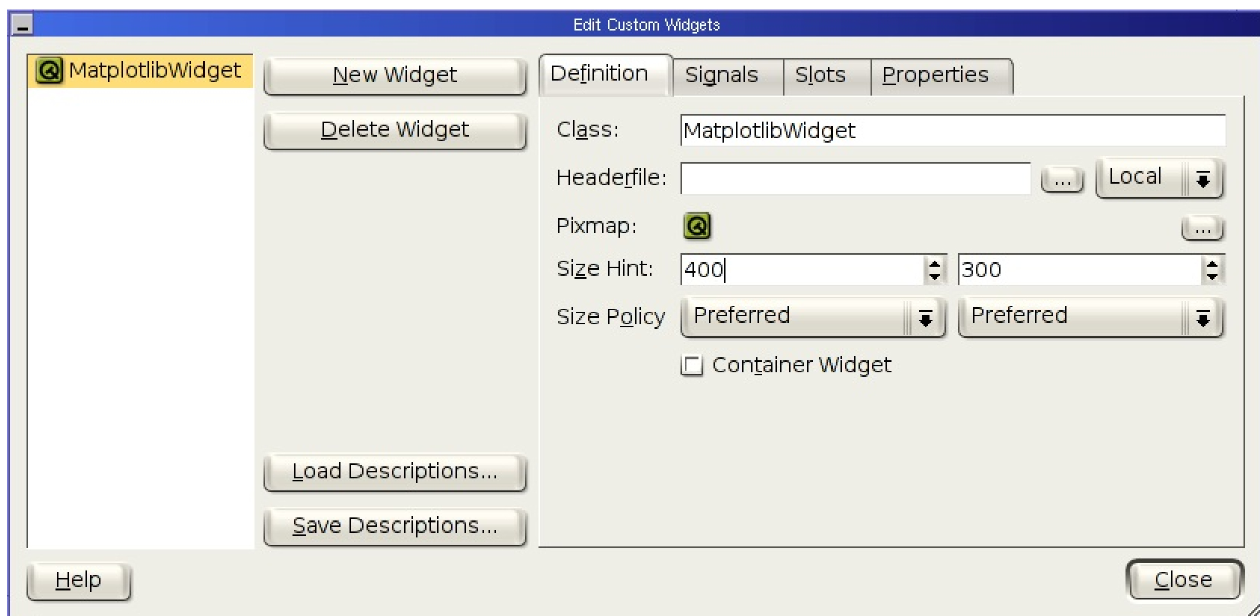
When the wizard appears, remove all of the menus and toolbars that it suggests to generate:



Now add a custom widget to your project as:

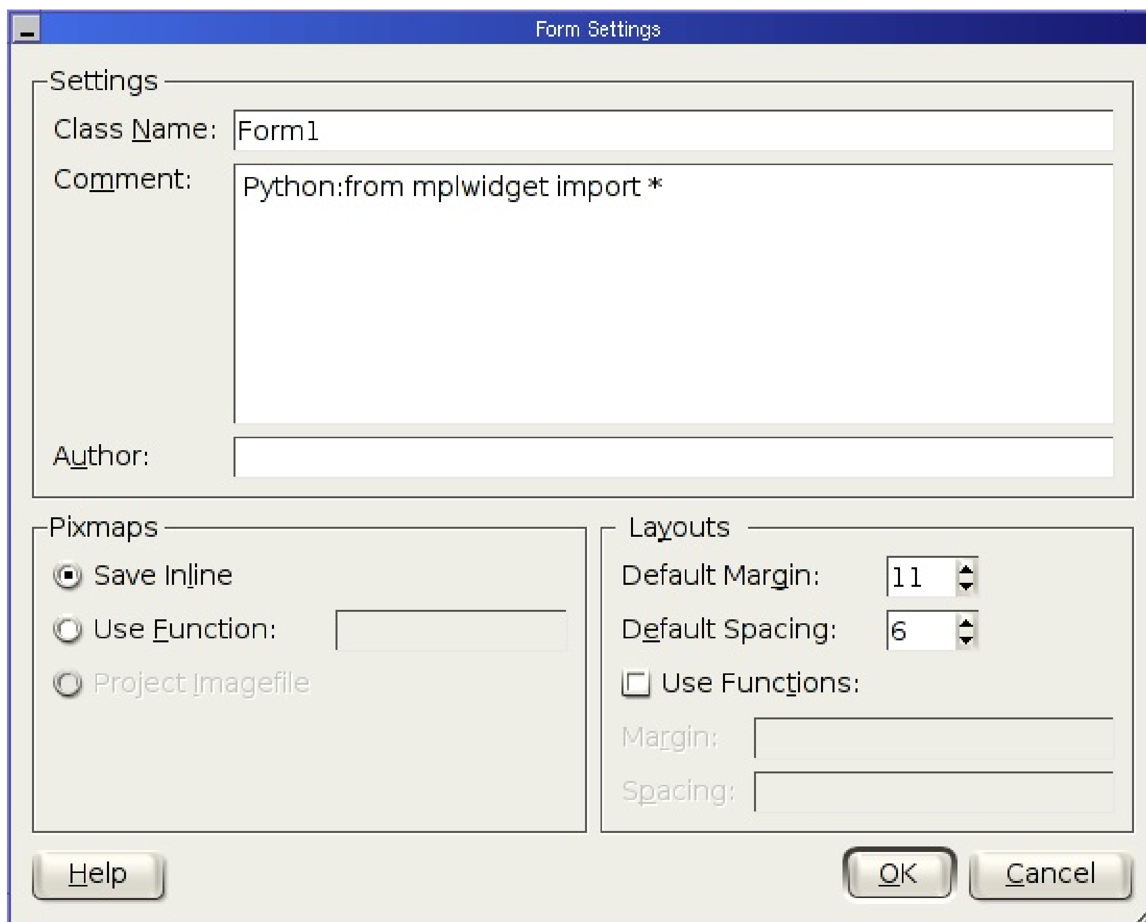


Give this new widget the name of and set the size to 400, 300:



The should now appear in your toolbox. Click on it and then click in the Form that has been created. Also perform “Lay Out Vertically (Ctrl+L)” and “Adjust Size (Ctrl+J)” operations on the Form (without the widget being selected) if you know what those are. At this point your workspace might look something like this:

Now we need to get the “import” setting put in, as specified here:



Of course, you'll also need the file that will then be included: `mplwidget.py` .

The product of all of these operations in Designer is a .ui file. So, save the Form we've been working on and call it "mplwidget_tutorial.ui".

One of the things that can mess this up is that Designer automatically increments the names given to widget instances (Form1, Form2, matplotlibWidget1, matplotlibWidget2, etc.) so if those don't match up with what I used, you may need to make some logical adjustments to your procedure.

Now, we use the tool pyuic (included with PyQt to convert this .ui file into a Python class. This is easily accomplished with:

```
pyuic mplwidget_tutorial.ui > mplwidget_tutorial.py
```

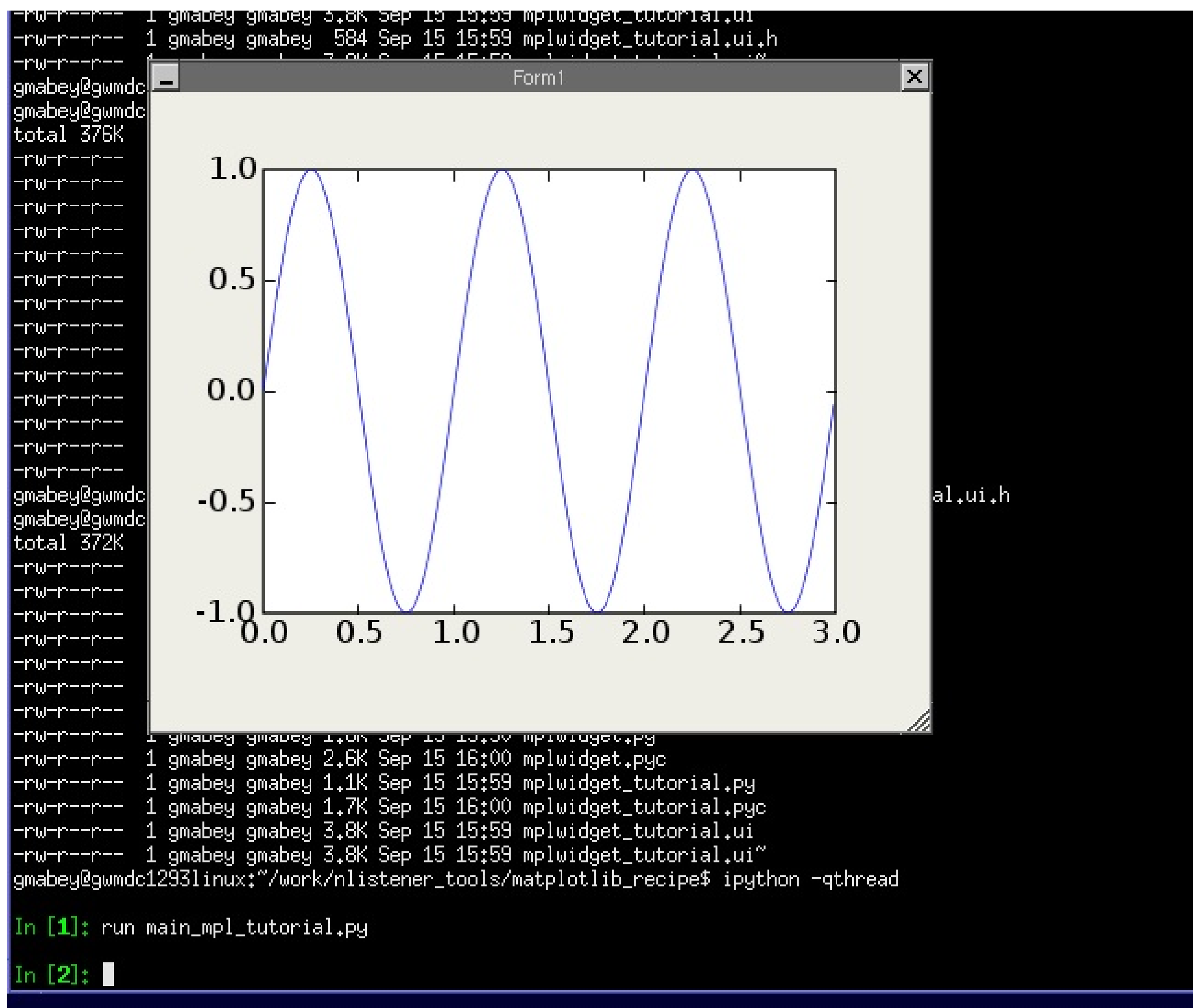
Go ahead and view the contents of mplwidget_tutorial.py and compare it with what I got. You can also look)# at my `mplwidget_tutorial.ui`

Now, it's nice to have the invocation written up in its own `main_mpl_tutorial.py` file, which is amazingly short:)#

```
from mplwidget_tutorial import *

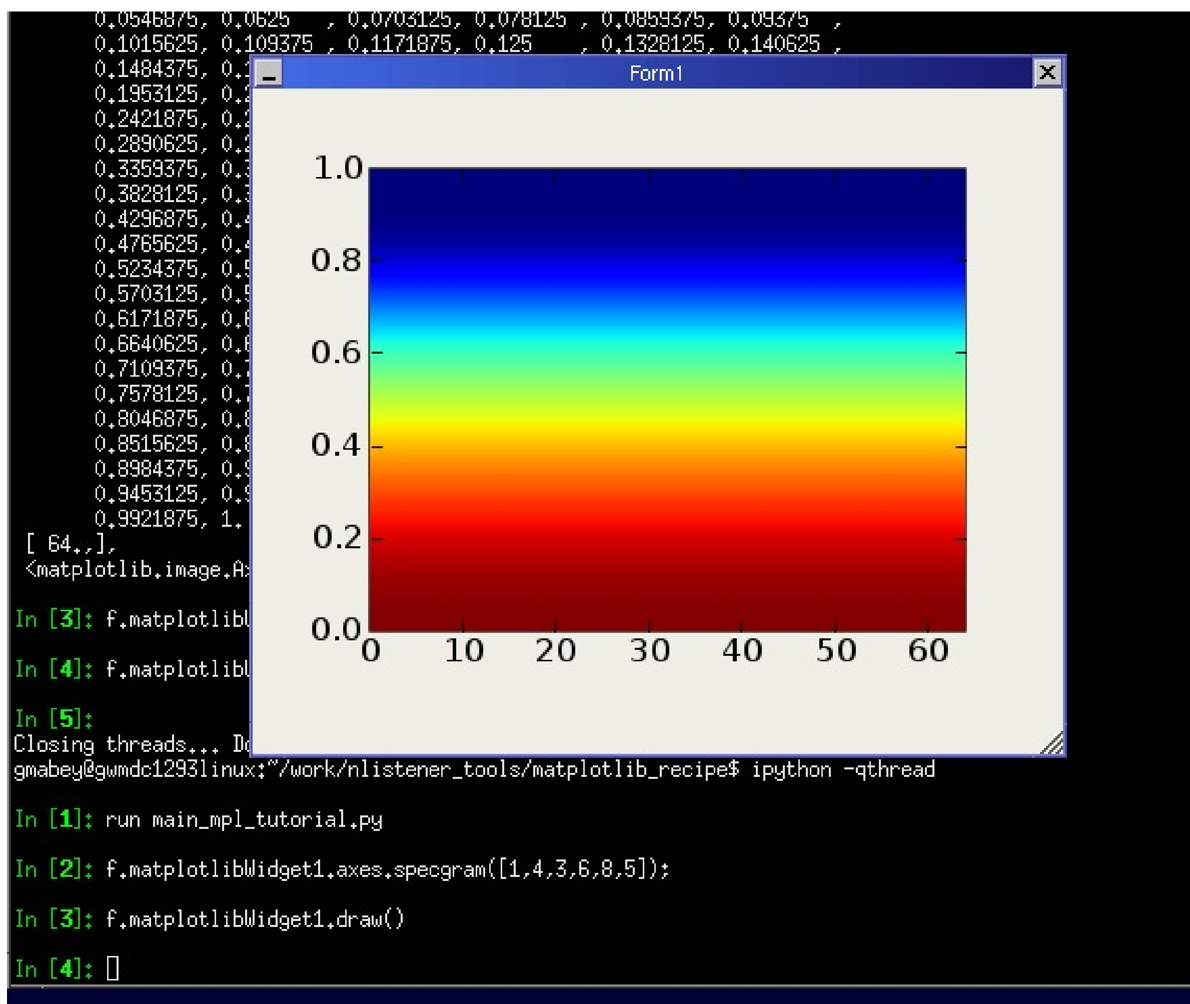
f = Form1()
f.show()
```

Then, we want to start ipython and instantiate the window. In order to succeed in this endeavor, there are a bunch of packages (each with minimum version requirements) you'll need, like python2.3-ipython, python2.3-qt3, and so on. The most important one, though, is ipython >= 0.6.13 (I think). From that version on, there is a super-great feature that adds an invocation switch {{{-qthread}}} that starts a QApplication in a separate thread, in such a way that the ipython prompt can still interact with it.



So, if the above command doesn't work for you, check the version. It may also be invoked as “python2.3-ipython” depending on your configuration.

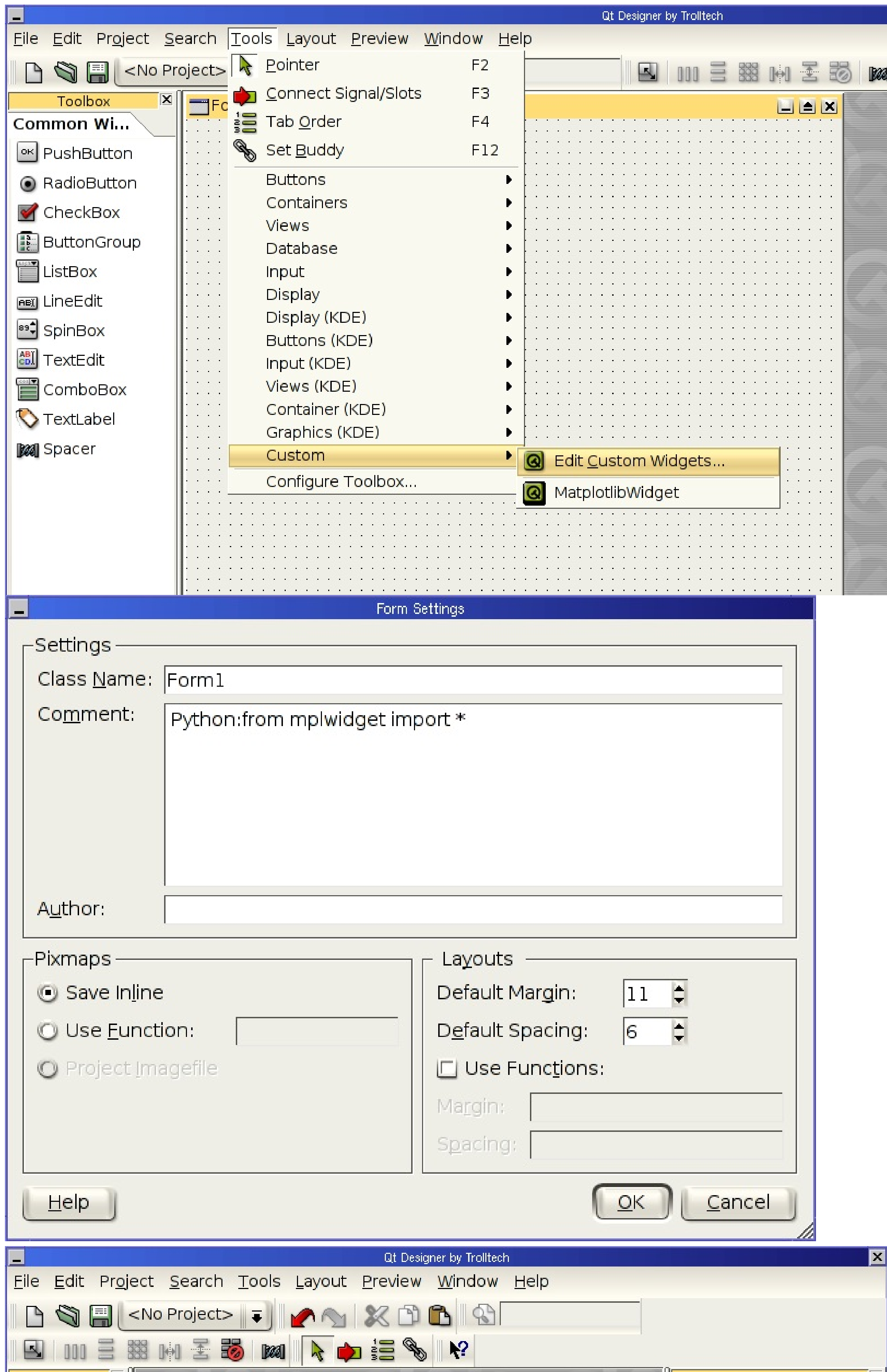
Okay, now comes the coolest part: the interaction.

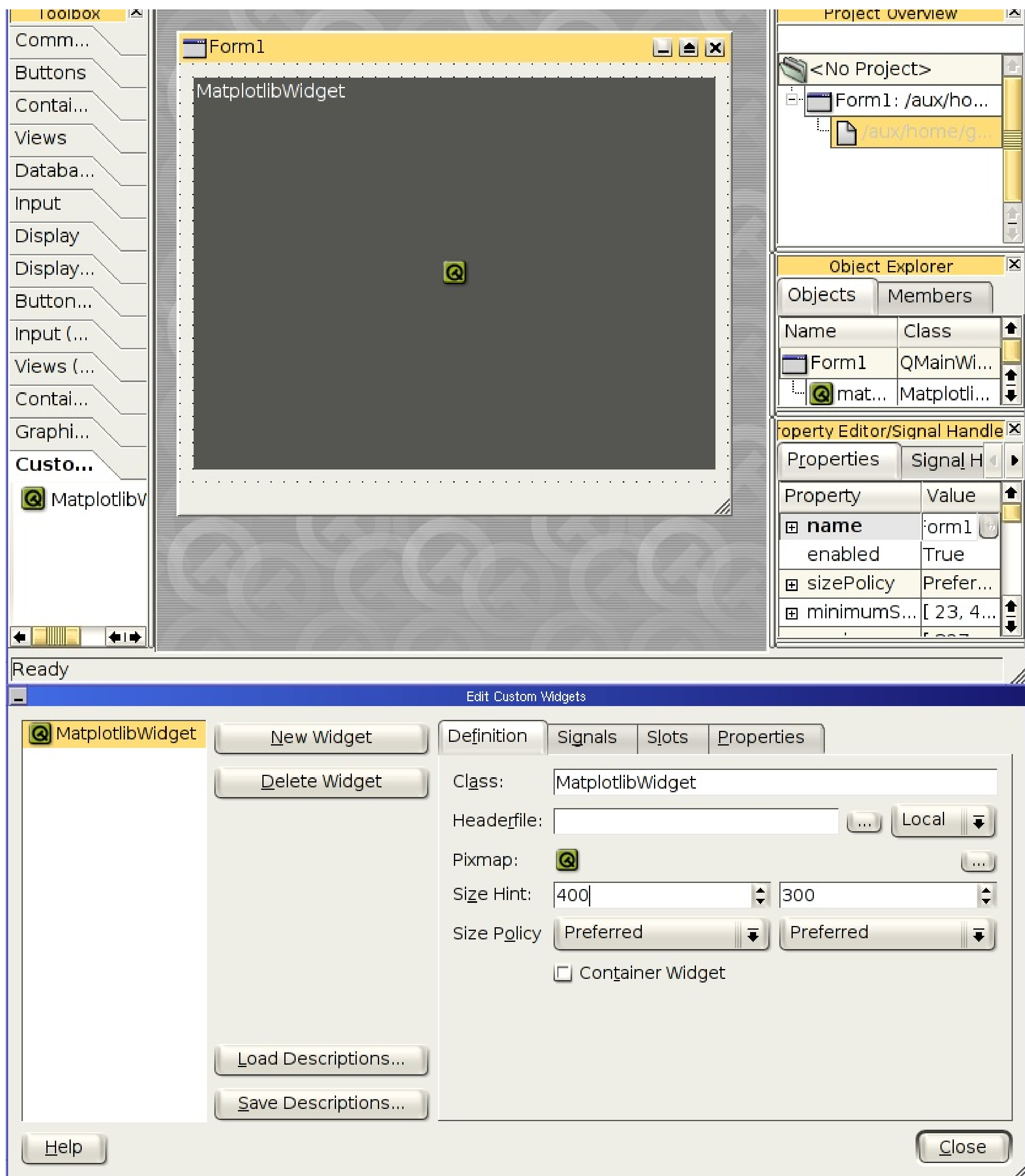


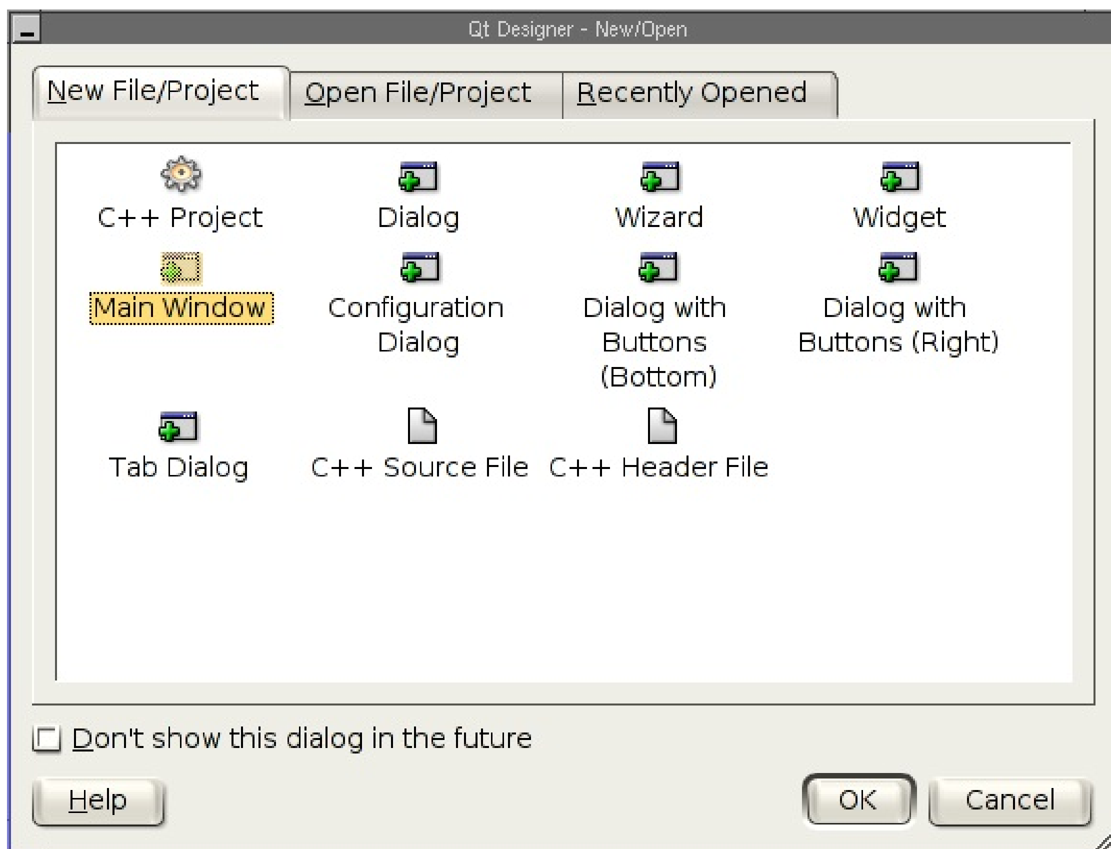
And, it is very easy to go back to Designer, add a button, re-run pyuic, and you've got another version.

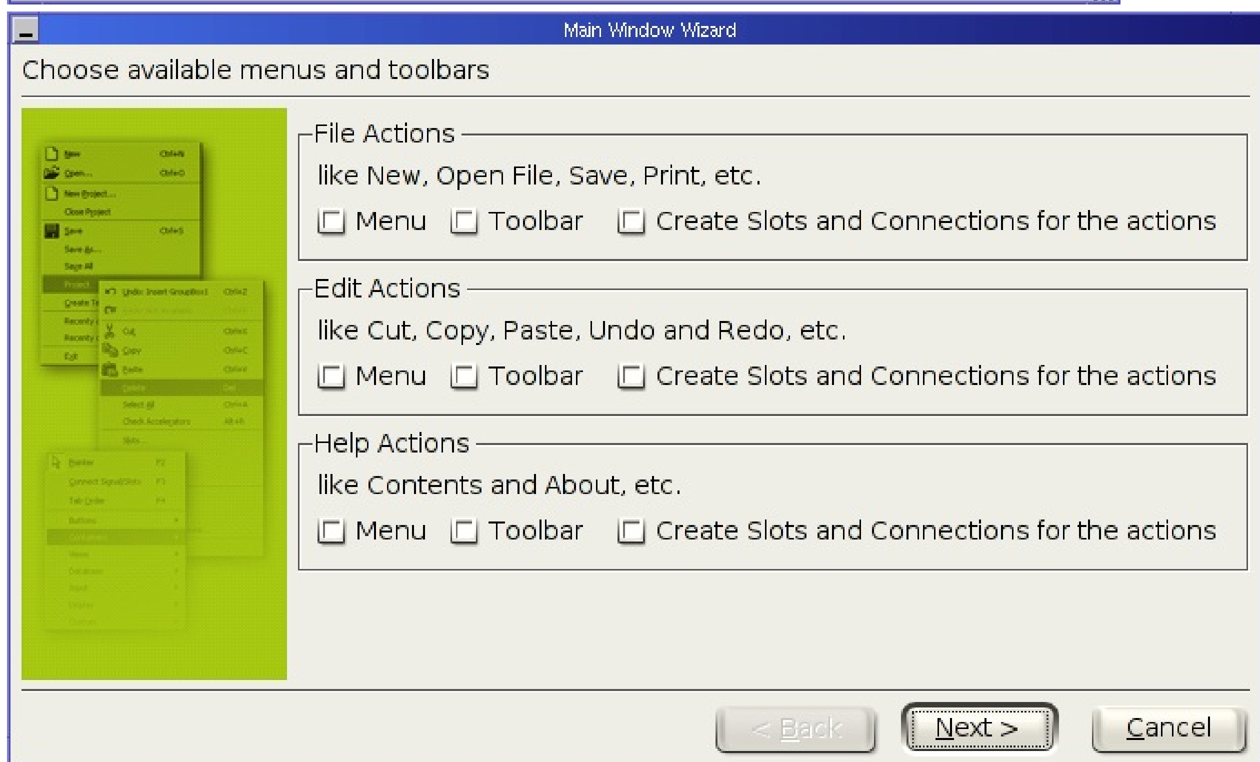
Attachments

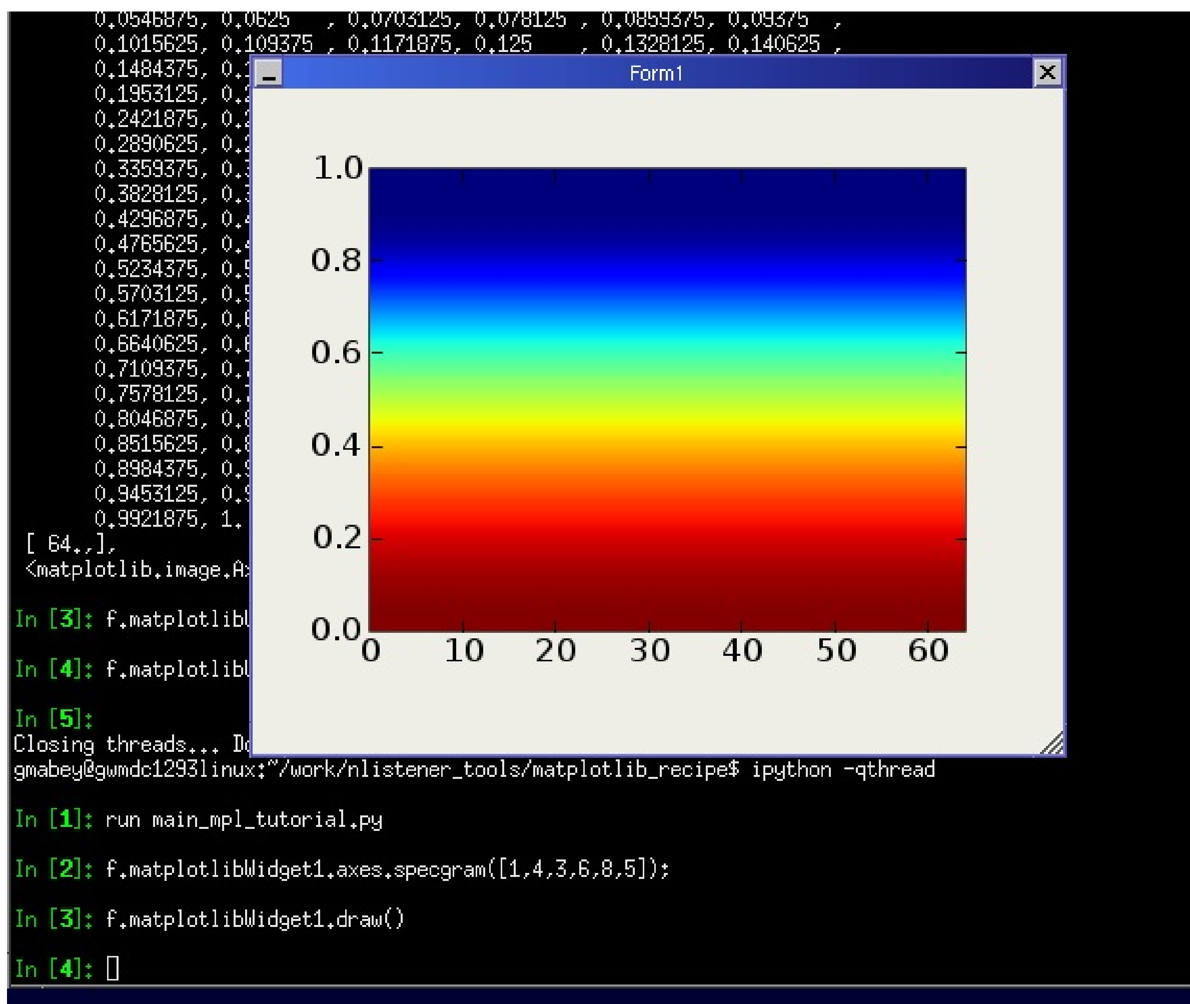
- `designer_edit_custom_widgets.png`
- `designer_form_settings_comment.png`
- `designer_full_workspace.png`
- `designer_new_widget.png`
- `designer_newopen.png`
- `designer_wizard.png`
- `ipython_interacted.png`
- `ipython_invoked.png`
- `main_mpl_tutorial.py`
- `mplwidget.py`
- `mplwidget_tutorial.py`
- `mplwidget_tutorial.ui`


Qt Designer by Trolltech
File Edit Project Search Tools Layout Preview Window Help
Pointer F2
Connect Signal/Slots F3
Tab Order F4
Set Buddy F12
Buttons
Containers
Views
Database
Input
Display
Display (KDE)
Buttons (KDE)
Input (KDE)
Views (KDE)
Container (KDE)
Graphics (KDE)
Custom
Configure Toolbox...
Edit Custom Widgets...
MatplotlibWidget
Toolbox
Common Wi...
PushButton
RadioButton
CheckBox
ButtonGroup
ListBox
LineEdit
SpinBox
TextEdit
ComboBox
TextLabel
Spacer
Form Settings
Settings
Class Name: Form1
Comment: Python:from mplwidget import *
Author:
Pixmaps
Save Inline
Use Function:
Project Imagefile
Layouts
Default Margin: 11
Default Spacing: 6
Use Functions:
Margin:
Spacing:
Help
OK
Cancel
Qt Designer by Trolltech
File Edit Project Search Tools Layout Preview Window Help
<No Project>

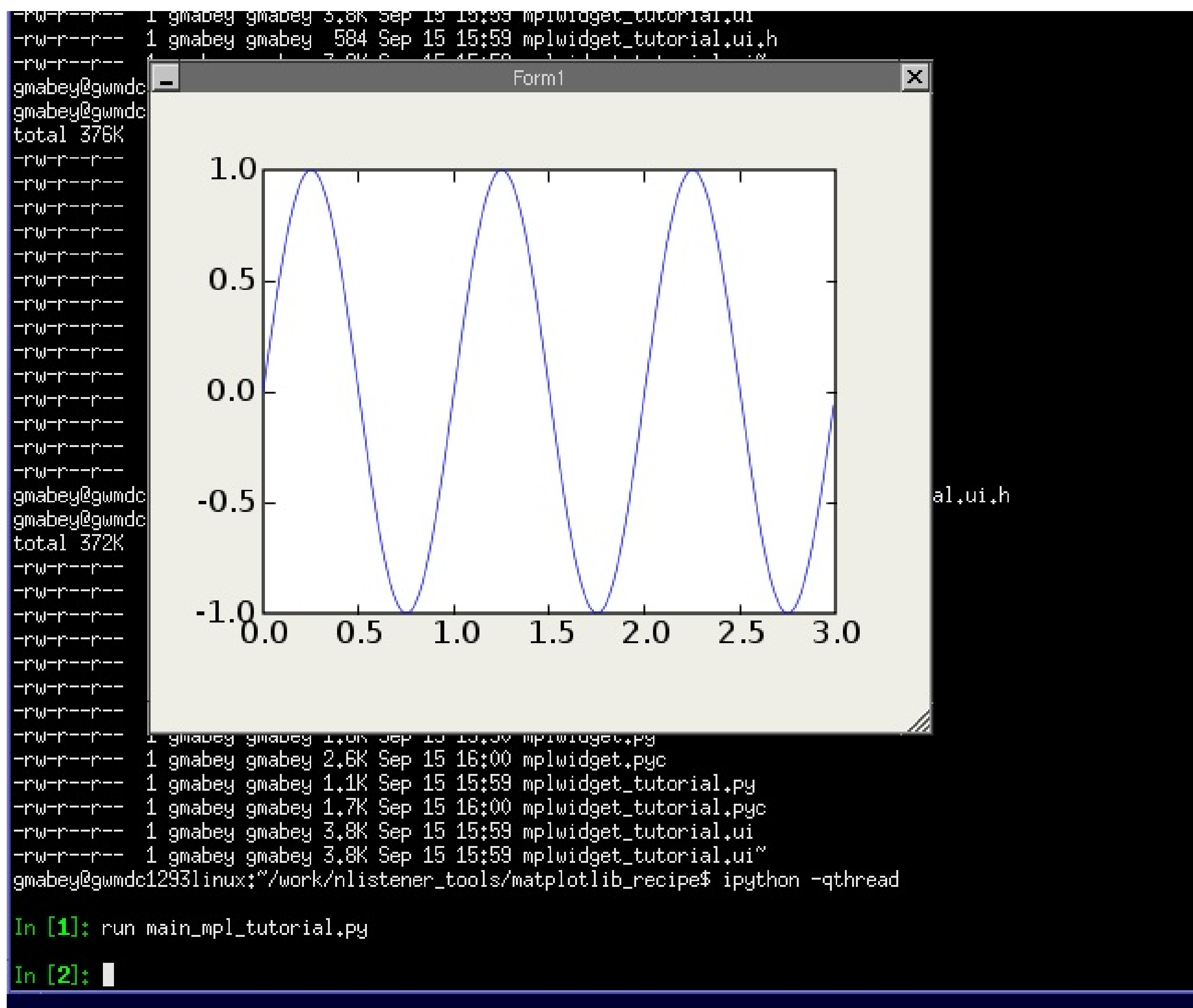

Toolbox
Comm...
Buttons
Contai...
Views
Databa...
Input
Display
Display...
Button...
Input (...
Views (...
Contai...
Graphi...
Custo...
MatplotlibV
Form1
MatplotlibWidget
Project Overview
<No Project>
Form1: /aux/ho...
/aux/home/g...
Object Explorer
Objects
Members
Name
Class
Form1
QMainWi...
mat...
Matplotli...
operty Editor/Signal Handle
Properties
Signal H
Property
Value
name
orm1
enabled
True
sizePolicy
Prefer...
minimumS...
[23, 4...
Ready
Edit Custom Widgets
MatplotlibWidget
New Widget
Delete Widget
Definition
Signals
Slots
Properties
Class:
MatplotlibWidget
Headerfile:
Local
Pixmap:
Size Hint:
400
300
Size Policy
Preferred
Preferred
Container Widget
Load Descriptions...
Save Descriptions...
Help
Close


Qt Designer - New/Open
New File/Project
Open File/Project
Recently Opened
C++ Project
Dialog
Wizard
Widget
Main Window
Configuration Dialog
Dialog with Buttons (Bottom)
Dialog with Buttons (Right)
Tab Dialog
C++ Source File
C++ Header File
Don't show this dialog in the future
Help
OK
Cancel


Main Window Wizard
Choose available menus and toolbars
File Actions
like New, Open File, Save, Print, etc.
Menu
Toolbar
Create Slots and Connections for the actions
Edit Actions
like Cut, Copy, Paste, Undo and Redo, etc.
Menu
Toolbar
Create Slots and Connections for the actions
Help Actions
like Contents and About, etc.
Menu
Toolbar
Create Slots and Connections for the actions
< Back
Next >
Cancel



0.1015625, 0.109375 , 0.1171875, 0.125 , 0.1328125, 0.140625 ,
Form1
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60
[64.,],
In [3]: f.matplotlibl
In [4]: f.matplotlibl
In [5]:
gmabey@gwmdc1293linux:~/work/nlistener_tools/matplotlib_recipe$ ipython -qthread
In [1]: run main_mpl_tutorial.py
In [2]: f.matplotlibWidget1.axes.specgram([1,4,3,6,8,5]);
In [3]: f.matplotlibWidget1.draw()
In [4]:


-rw-r--r-- 1 gmabey gmabey 584 Sep 15 15:59 mplwidget_tutorial.ui.h
total 376K
Form1
1.0
0.5
0.0
-0.5
-1.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
total 372K
-rw-r--r-- 1 gmabey gmabey 2.6K Sep 15 16:00 mplwidget.pyc
-rw-r--r-- 1 gmabey gmabey 1.1K Sep 15 15:59 mplwidget_tutorial.py
-rw-r--r-- 1 gmabey gmabey 1.7K Sep 15 16:00 mplwidget_tutorial.pyc
-rw-r--r-- 1 gmabey gmabey 3.8K Sep 15 15:59 mplwidget_tutorial.ui
-rw-r--r-- 1 gmabey gmabey 3.8K Sep 15 15:59 mplwidget_tutorial.ui~
gmabey@gwmdc1293linux:~/work/nlistener_tools/matplotlib_recipe$ ipython -qthread
In [1]: run main_mpl_tutorial.py
In [2]:

Matplotlib: using matplotlib in a CGI script

Trying to use matplotlib in a python CGI script naïvely will most likely result in the following error:

```
...
352, in _get_configdir
raise RuntimeError("'%s' is not a writable dir; you must set
environment variable HOME to be a writable dir "%h)
RuntimeError: '<WebServer DocumentRoot>' is not a writable dir; you
environment variable HOME to be a writable dir
```

Matplotlib needs the environment variable HOME to point to a writable directory. One way to accomplish this is to set this environment variable from within the CGI script on runtime (another way would be to modify the file but that would be not as portable). The following template can be used for a cgi that uses matplotlib to create a png image:

```
#!/usr/bin/python
import os,sys
import cgi
import cgitb; cgitb.enable()

# set HOME environment variable to a directory the httpd server can
os.environ[ 'HOME' ] = '/tmp/'

import matplotlib
# chose a non-GUI backend
matplotlib.use( 'Agg' )

import pylab

#Deals with inputing data into python from the html form
form = cgi.FieldStorage()

# construct your plot
pylab.plot([1,2,3])

print "Content-Type: image/png\n"

# save the plot as a png and output directly to webserver
pylab.savefig( sys.stdout, format='png' )
```

This image can then be accessed with a URL such as: http://localhost/showpng.py

As documented,some backends will not allow the output to be sent to sys.stdout. It is possible to replace the last line with the following to work around this:

```
pylab.savefig( "tempfile.png", format='png' )
import shutil
shutil.copyfileobj(open("tempfile.png",'rb'), sys.stdout)
```

(Of course it is necessary to create and delete proper temp files to use this in production.)

Matplotlib / Pseudo Color Plots

Matplotlib: colormap transformations

Operating on color vectors

Ever wanted to reverse a colormap, or to desaturate one ? Here is a routine to apply a function to the look up table of a colormap:

```
def cmap_map(function,cmap):
    """ Applies function (which should operate on vectors of shape
 [r, g, b], on colormap cmap. This routine will break any discontin
 """
    cdict = cmap._segmentdata
    step_dict = {}
    # Firt get the list of points where the segments start or end
    for key in ('red','green','blue'):         step_dict[key] = map
    step_list = sum(step_dict.values(), [])
    step_list = array(list(set(step_list)))
    # Then compute the LUT, and apply the function to the LUT
    reduced_cmap = lambda step : array(cmap(step)[0:3])
    old_LUT = array(map( reduced_cmap, step_list))
    new_LUT = array(map( function, old_LUT))
    # Now try to make a minimal segment definition of the new LUT
    cdict = {}
    for i,key in enumerate(('red','green','blue')):
        this_cdict = {}
        for j,step in enumerate(step_list):
            if step in step_dict[key]:
                this_cdict[step] = new_LUT[j,i]
            elif new_LUT[j,i]!=old_LUT[j,i]:
                this_cdict[step] = new_LUT[j,i]
        colorvector=  map(lambda x: x + (x[1], ), this_cdict.items(
        colorvector.sort()
        cdict[key] = colorvector

    return matplotlib.colors.LinearSegmentedColormap('colormap',cdi
```

Lets try it out: I want a jet colormap, but lighter, so that I can plot things on top of it:

```
light_jet = cmap_map(lambda x: x/2+0.5, cm.jet)
x,y=mgrid[1:2,1:10:0.1]
imshow(y, cmap=light_jet)
```

[](files/attachments/Matplotlib_ColormapTransformations/light_jet4.png

As a comparison, this is what the original jet looks like: []
(files/attachments/Matplotlib_ColormapTransformations/jet.png

Operating on indices

OK, but what if you want to change the indices of a colormap, but not its colors.

```
def cmap_xmap(function,cmap):
    """ Applies function, on the indices of colormap cmap. Beware, 
    should map the [0, 1] segment to itself, or you are in for surp

    See also cmap_xmap.
    """
    cdict = cmap._segmentdata
    function_to_map = lambda x : (function(x[0]), x[1], x[2])
    for key in ('red','green','blue'):         cdict[key] = map(fur
        cdict[key].sort()
        assert (cdict[key][0]<0 or cdict[key][-1]>1), "Resulting ir

    return matplotlib.colors.LinearSegmentedColormap('colormap',cdi
```

Discrete colormap

Here is how you can discretize a continuous colormap.

```
def cmap_discretize(cmap, N):
    """Return a discrete colormap from the continuous colormap cmap

 cmap: colormap instance, eg. cm.jet.
 N: number of colors.

 Example
 x = resize(arange(100), (5,100))
 djet = cmap_discretize(cm.jet, 5)
 imshow(x, cmap=djet)
 """

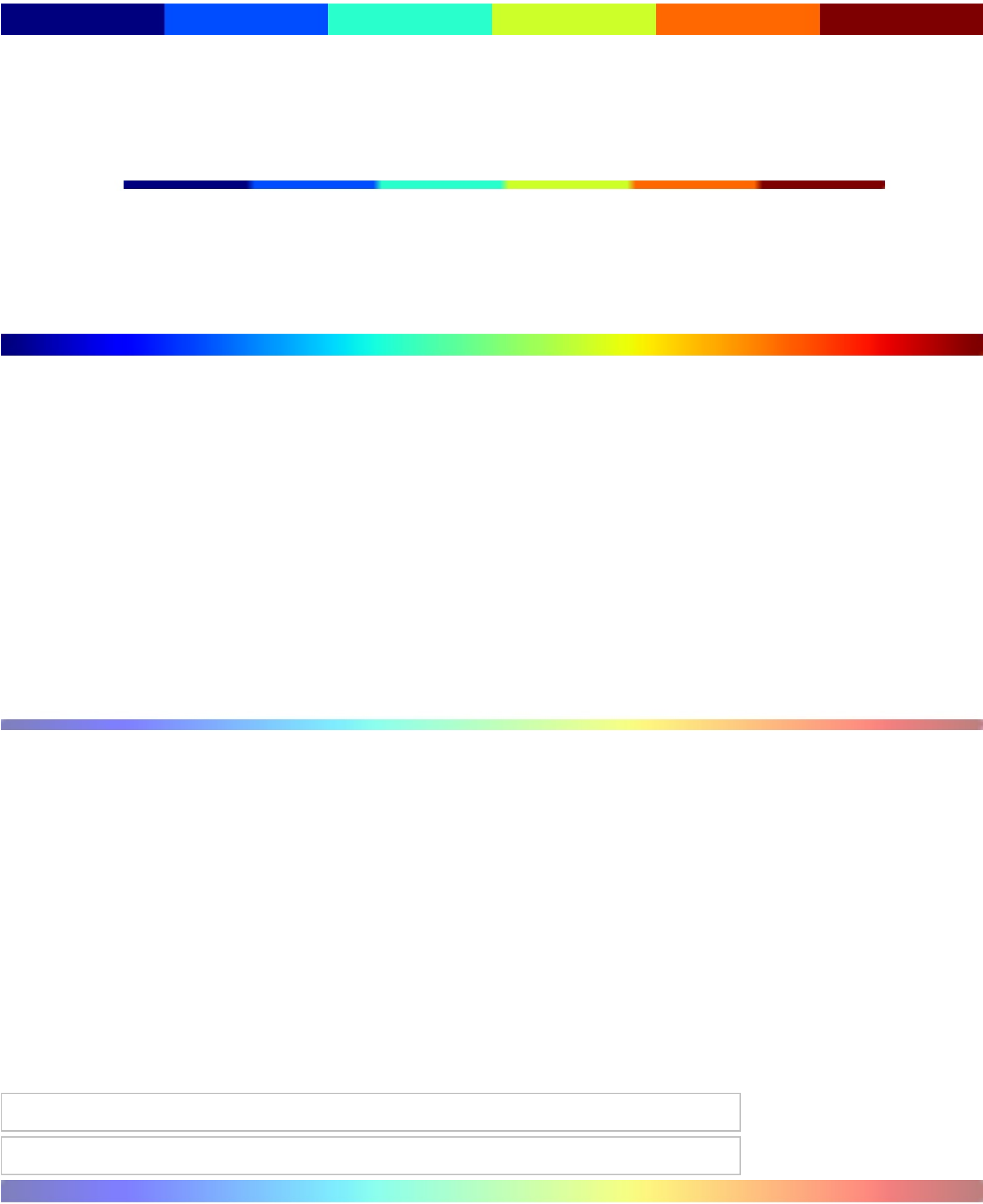
    if type(cmap) == str:
        cmap = get_cmap(cmap)
    colors_i = concatenate((linspace(0, 1., N), (0.,0.,0.,0.)))
    colors_rgba = cmap(colors_i)
    indices = linspace(0, 1., N+1)
    cdict = {}
    for ki,key in enumerate(('red','green','blue')):
        cdict[key] = [ (indices[i], colors_rgba[i-1,ki], colors_rgb
    # Return colormap object.
    return matplotlib.colors.LinearSegmentedColormap(cmap.name + "_
```

So for instance, this is what you would get by doing {{{cmap_discretize(cm.jet, 6)}}}.



Attachments

- dicrete_jet1.png
- discrete_jet.png
- jet.png
- light_jet.png
- light_jet2.png
- light_jet3.png
- light_jet4.png



Matplotlib: converting a matrix to a raster image

Scipy provides a command (imsave) to make a raster (png, jpg...) image from a 2D array, each pixel corresponding to one value of the array. Yet the image is black and white.

Here is a recipy to do this with Matplotlib, and use a colormap to give color to the image.

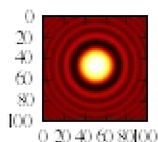
```
from matplotlib.pyplot import *
from scipy import mgrid

def imsave(filename, X, **kwargs):
    """ Homebrewed imsave to have nice colors... """
    figsize=(array(X.shape)/100.0)[::-1]
    rcParams.update({'figure.figsize':figsize})
    fig = figure(figsize=figsize)
    axes([0,0,1,1]) # Make the plot occupy the whole canvas
    axis('off')
    fig.set_size_inches(figsize)
    imshow(X,origin='lower', **kwargs)
    savefig(filename, facecolor='black', edgecolor='black', dpi=100
    close(fig)

X,Y=mgrid[-5:5:0.1,-5:5:0.1]
Z=sin(X**2+Y**2+1e-4)/(X**2+Y**2+1e-4) # Create the data to be plot
imsave('imsave.png', Z, cmap=cm.hot )
```

```
imshow(imread('imsave.png'))
```

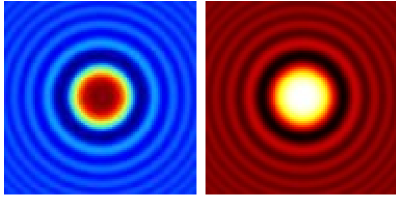
```
<matplotlib.image.AxesImage at 0x7f1733edf610>
```



Attachments

- `imsave.jpg`

- `imsave.png`



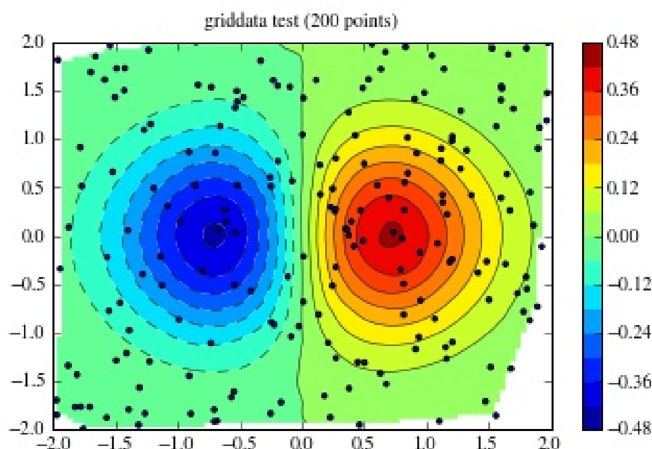
Matplotlib: gridding irregularly spaced data

A commonly asked question on the matplotlib mailing lists is “how do I make a contour plot of my irregularly spaced data?”. The answer is, first you interpolate it to a regular grid. As of version 0.98.3, matplotlib provides a griddata function that behaves similarly to the matlab version. It performs “natural neighbor interpolation” of irregularly spaced data a regular grid, which you can then plot with contour, imshow or pcolor.

Example 1

This requires Scipy 0.9:

```
import numpy as np
from scipy.interpolate import griddata
import matplotlib.pyplot as plt
import numpy.ma as ma
from numpy.random import uniform, seed
# make up some randomly distributed data
seed(1234)
npts = 200
x = uniform(-2,2,npts)
y = uniform(-2,2,npts)
z = x*np.exp(-x**2-y**2)
# define grid.
xi = np.linspace(-2.1,2.1,100)
yi = np.linspace(-2.1,2.1,100)
# grid the data.
zi = griddata((x, y), z, (xi[None,:], yi[:,None]), method='cubic')
# contour the gridded data, plotting dots at the randomly spaced da
CS = plt.contour(xi,yi,zi,15,linewidths=0.5,colors='k')
CS = plt.contourf(xi,yi,zi,15,cmap=plt.cm.jet)
plt.colorbar() # draw colorbar
# plot data points.
plt.scatter(x,y,marker='o',c='b',s=5)
plt.xlim(-2,2)
plt.ylim(-2,2)
plt.title('griddata test (%d points)' % npts)
plt.show()
```



Example 2

```
import numpy as np
from matplotlib.mlab import griddata
import matplotlib.pyplot as plt
import numpy.ma as ma
from numpy.random import uniform
# make up some randomly distributed data
npts = 200
x = uniform(-2,2,npts)
y = uniform(-2,2,npts)
z = x*np.exp(-x**2-y**2)
# define grid.
xi = np.linspace(-2.1,2.1,100)
yi = np.linspace(-2.1,2.1,100)
# grid the data.
zi = griddata(x,y,z,xi,yi)
# contour the gridded data, plotting dots at the randomly spaced da
CS = plt.contour(xi,yi,zi,15,linewidths=0.5,colors='k')
CS = plt.contourf(xi,yi,zi,15,cmap=plt.cm.jet)
plt.colorbar() # draw colorbar
# plot data points.
plt.scatter(x,y,marker='o',c='b',s=5)
plt.xlim(-2,2)
plt.ylim(-2,2)
plt.title('griddata test (%d points)' % npts)
plt.show()
```

By default, griddata uses the scikits delaunay package (included in matplotlib) to do the natural neighbor interpolation. Unfortunately, the delaunay package is known to fail for some nearly pathological cases. If you run into one of those cases, you can install the matplotlib natgrid toolkit. Once that is installed, the

griddata function will use it instead of delaunay to do the interpolation. The natgrid algorithm is a bit more robust, but cannot be included in matplotlib proper because of licensing issues.

The radial basis function module in the scipy sandbox can also be used to interpolate/smooth scattered data in n dimensions. See [“Cookbook/RadialBasisFunctions”] for details.

Example 3

A less robust but perhaps more intuitive method is presented in the code below. This function takes three 1D arrays, namely two independent data arrays and one dependent data array and bins them into a 2D grid. On top of that, the code also returns two other grids, one where each binned value represents the number of points in that bin and another in which each bin contains the indices of the original dependent array which are contained in that bin. These can be further used for interpolation between bins if necessary.

The is essentially an Occam’s Razor approach to the matplotlib.mlab griddata function, as both produce similar results.

```
# griddata.py - 2010-07-11 ccampo
import numpy as np

def griddata(x, y, z, binsize=0.01, retbin=True, retloc=True):
    """
 Place unevenly spaced 2D data on a grid by 2D binning (nearest
 neighbor interpolation).

 Parameters
 ----------
 x : ndarray (1D)
 The idependent data x-axis of the grid.
 y : ndarray (1D)
 The idependent data y-axis of the grid.
 z : ndarray (1D)
 The dependent data in the form z = f(x,y).
 binsize : scalar, optional
 The full width and height of each bin on the grid.  If each
 bin is a cube, then this is the x and y dimension.  This is
 the step in both directions, x and y. Defaults to 0.01.
 retbin : boolean, optional
 Function returns `bins` variable (see below for description)
 if set to True.  Defaults to True.
 retloc : boolean, optional
 Function returns `wherebins` variable (see below for description)
 if set to True.  Defaults to True.

 Returns
 -------
```

```
grid : ndarray (2D)
The evenly gridded data.  The value of each cell is the median
value of the contents of the bin.
bins : ndarray (2D)
A grid the same shape as `grid`, except the value of each cell
is the number of points in that bin.  Returns only if
`retbin` is set to True.
wherebin : list (2D)
A 2D list the same shape as `grid` and `bins` where each cell
contains the indicies of `z` which contain the values stored
in the particular bin.

Revisions
---------
2010-07-11  ccampo  Initial version
"""
   # get extrema values.
   xmin, xmax = x.min(), x.max()
   ymin, ymax = y.min(), y.max()

   # make coordinate arrays.
   xi      = np.arange(xmin, xmax+binsize, binsize)
   yi      = np.arange(ymin, ymax+binsize, binsize)
   xi, yi = np.meshgrid(xi,yi)

   # make the grid.
   grid           = np.zeros(xi.shape, dtype=x.dtype)
   nrow, ncol = grid.shape
   if retbin: bins = np.copy(grid)

   # create list in same shape as grid to store indices
   if retloc:
       wherebin = np.copy(grid)
       wherebin = wherebin.tolist()

   # fill in the grid.
   for row in range(nrow):
       for col in range(ncol):
           xc = xi[row, col]    # x coordinate.
           yc = yi[row, col]    # y coordinate.

           # find the position that xc and yc correspond to.
           posx = np.abs(x - xc)
           posy = np.abs(y - yc)
           ibin = np.logical_and(posx < binsize/2., posy < binsize
           ind  = np.where(ibin == True)[0]

           # fill the bin.
           bin = z[ibin]
           if retloc: wherebin[row][col] = ind
           if retbin: bins[row, col] = bin.size
           if bin.size != 0:
               binval         = np.median(bin)
```

```
                grid[row, col] = binval
            else:
                grid[row, col] = np.nan   # fill empty bins with na

    # return the grid
    if retbin:
        if retloc:
            return grid, bins, wherebin
        else:
            return grid, bins
    else:
        if retloc:
            return grid, wherebin
        else:
            return grid
```

The following example demonstrates a usage of this method.

```
import numpy as np
import matplotlib.pyplot as plt
import griddata

npr = np.random
npts = 3000.                            # the total number of data 
x = npr.normal(size=npts)            # create some normally distrib
y = npr.normal(size=npts)            # ... do the same for y.
zorig = x**2 + y**2                      # z is a function of the f
noise = npr.normal(scale=1.0, size=npts) # add a good amount of noi
z = zorig + noise                    # z = f(x, y) = x**2 + y**2

# plot some profiles / cross-sections for some visualization.  our
# function is a symmetric, upward opening paraboloid z = x**2 + y**
# We expect it to be symmetric about and and y, attain a minimum on
# the origin and display minor Gaussian noise.

plt.ion()   # pyplot interactive mode on

# x vs z cross-section.  notice the noise.
plt.plot(x, z, '.')
plt.title('X vs Z=F(X,Y=constant)')
plt.xlabel('X')
plt.ylabel('Z')

# y vs z cross-section.  notice the noise.
plt.plot(y, z, '.')
plt.title('Y vs Z=F(Y,X=constant)')
plt.xlabel('Y')
plt.ylabel('Z')

# now show the dependent data (x vs y).  we could represent the z d
```

```
# as a third axis by either a 3d plot or contour plot, but we need
# grid it first....
plt.plot(x, y, '.')
plt.title('X vs Y')
plt.xlabel('X')
plt.ylabel('Y')

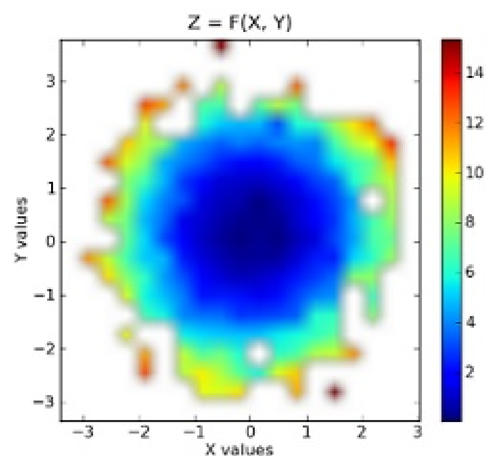
# enter the gridding.  imagine drawing a symmetrical grid over the
# plot above.  the binsize is the width and height of one of the gr
# cells, or bins in units of x and y.
binsize = 0.3
grid, bins, binloc = griddata.griddata(x, y, z, binsize=binsize)  #

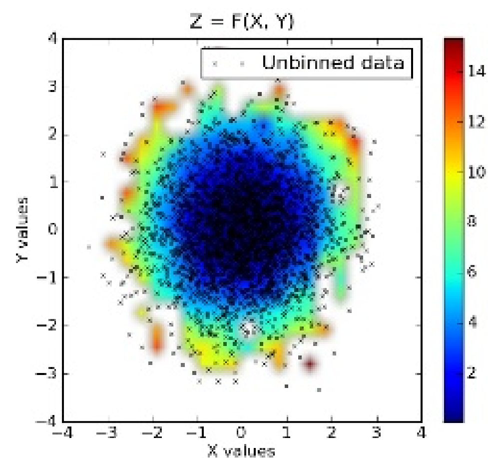
# minimum values for colorbar. filter our nans which are in the gri
zmin    = grid[np.where(np.isnan(grid) == False)].min()
zmax    = grid[np.where(np.isnan(grid) == False)].max()

# colorbar stuff
palette = plt.matplotlib.colors.LinearSegmentedColormap('jet3',plt.
palette.set_under(alpha=0.0)

# plot the results.  first plot is x, y vs z, where z is a filled l
extent = (x.min(), x.max(), y.min(), y.max()) # extent of the plot
plt.subplot(1, 2, 1)
plt.imshow(grid, extent=extent, cmap=palette, origin='lower', vmin=
plt.xlabel('X values')
plt.ylabel('Y values')
plt.title('Z = F(X, Y)')
plt.colorbar()

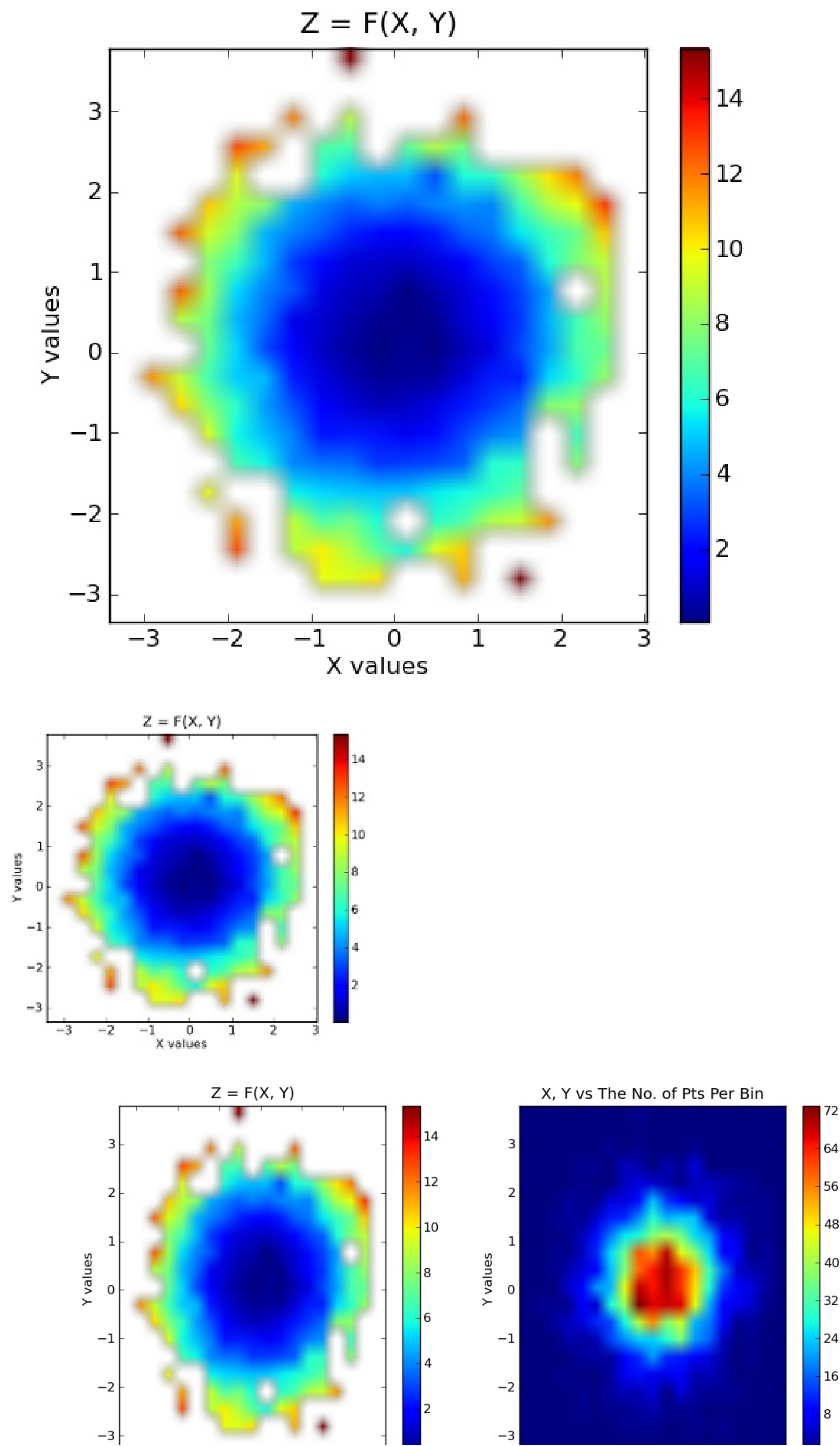
# now show the number of points in each bin.  since the independent
# Gaussian distributed, we expect a 2D Gaussian.
plt.subplot(1, 2, 2)
plt.imshow(bins, extent=extent, cmap=palette, origin='lower', vmin=
plt.xlabel('X values')
plt.ylabel('Y values')
plt.title('X, Y vs The No. of Pts Per Bin')
plt.colorbar()
```

The binned data: 

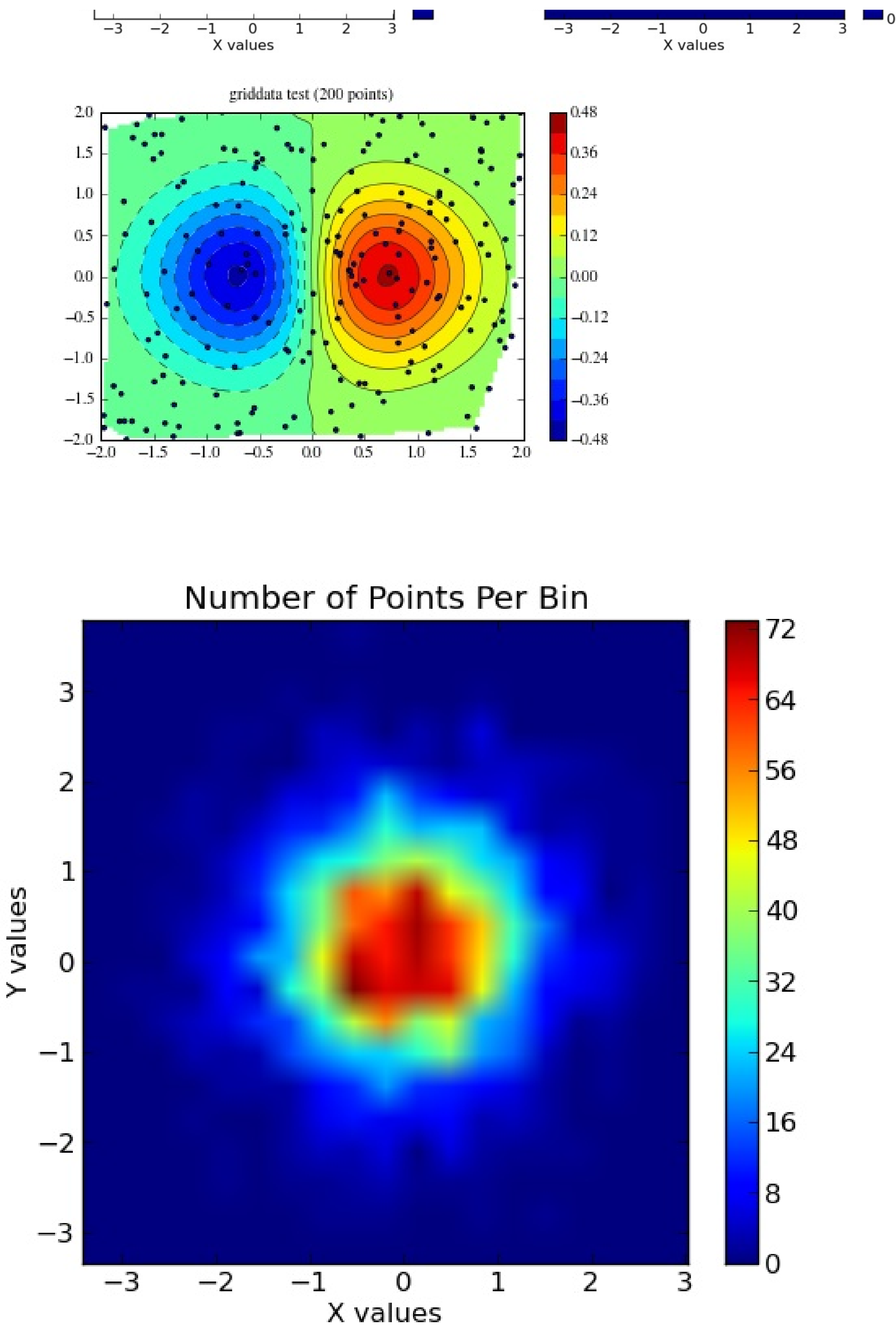
Raw data superimposed on top of binned data: 

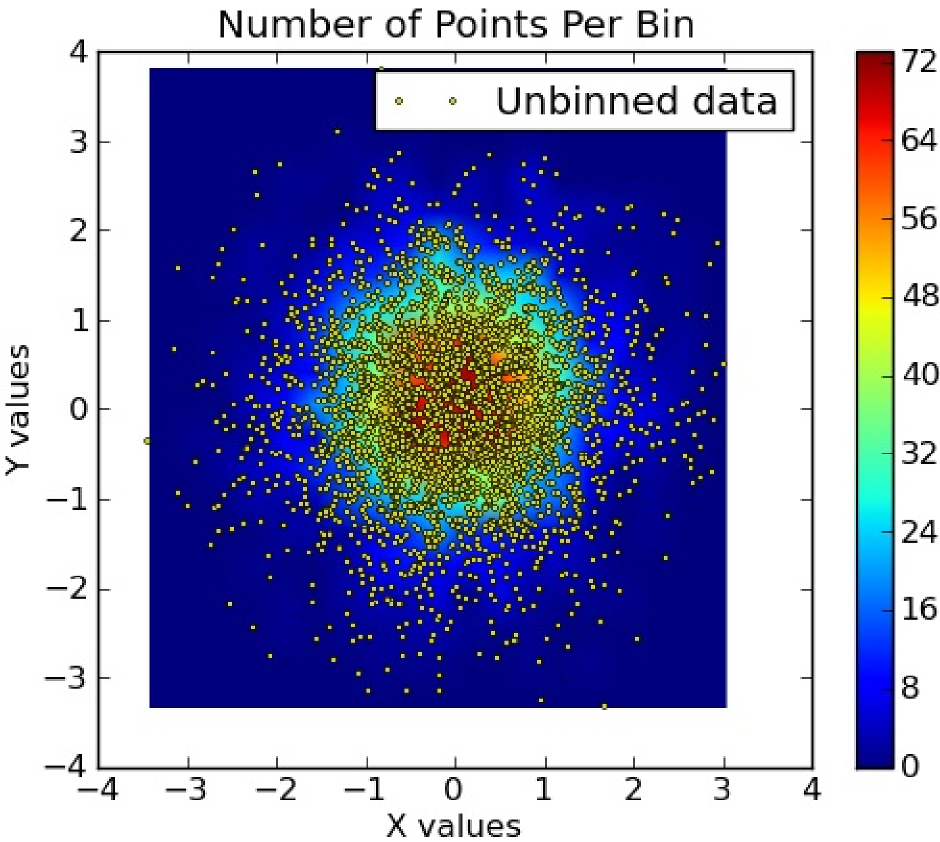
Attachments

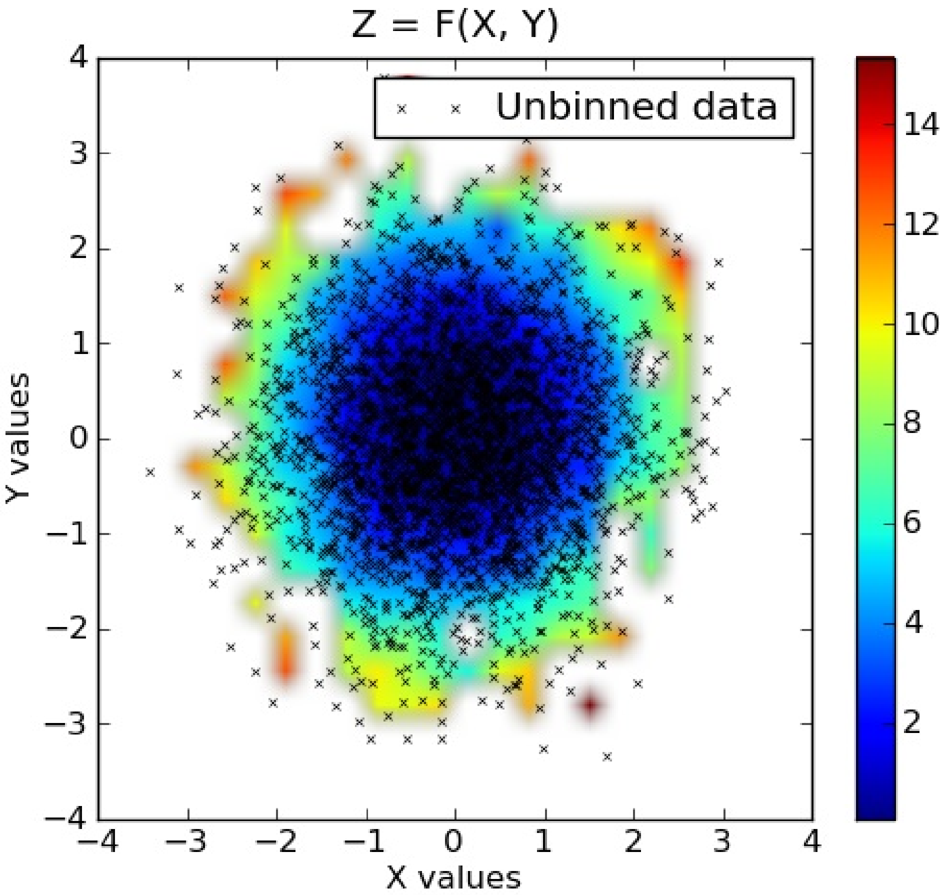
- `bin.png`
- `bin_small`
- `bin_small.png`
- `binned_data.png`
- `griddataexample1.png`
- `ppb.png`
- `ppb_raw.png`
- `raw.png`
- `raw_small`
- `raw_small.png`
- `unbinned_data.png`

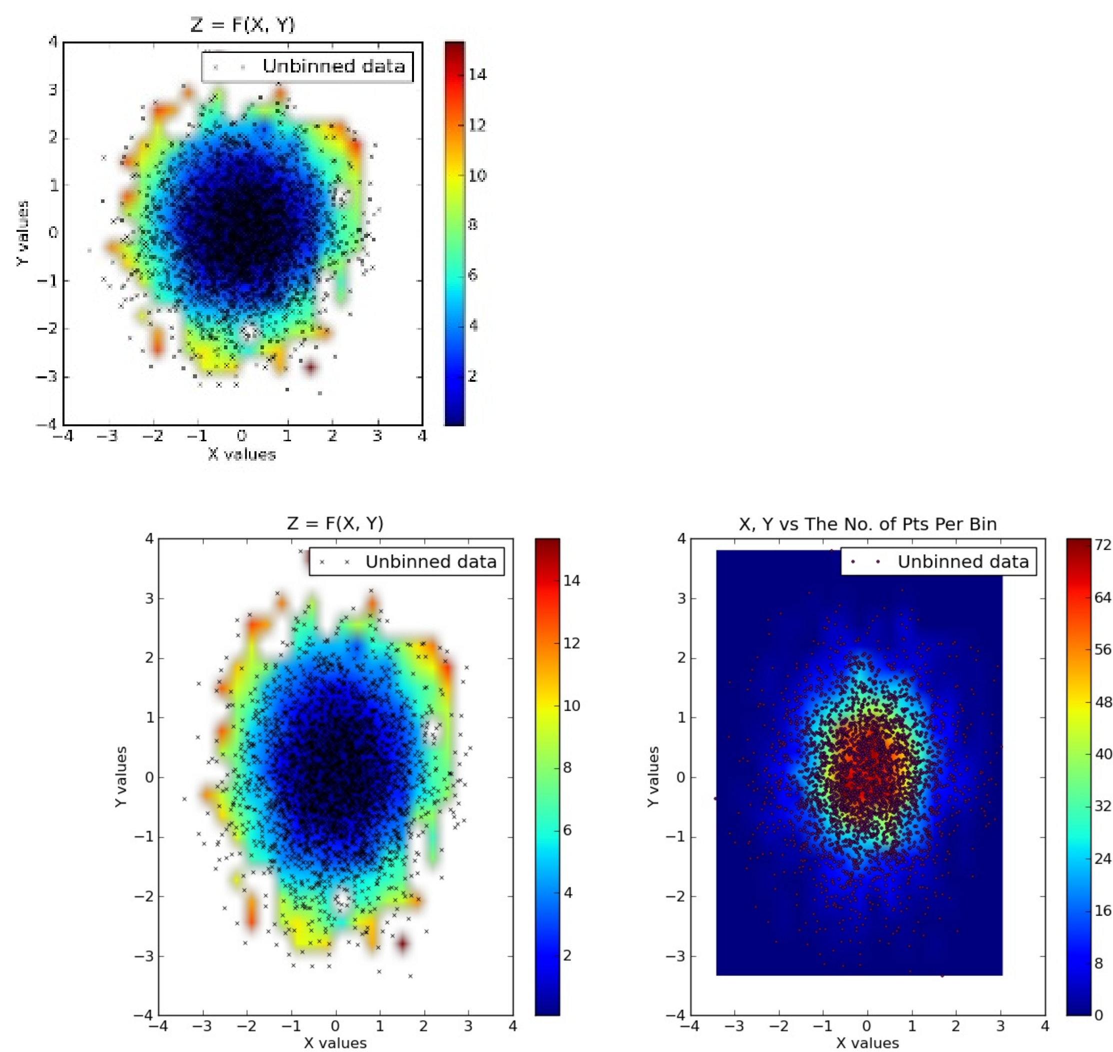


Z = F(X, Y)
Y values
X values
Z = F(X, Y)
Y values
X values
Z = F(X, Y)
Y values
X, Y vs The No. of Pts Per Bin
Y values


X values
X values
griddata test (200 points)
Number of Points Per Bin
Y values
X values


Number of Points Per Bin
Unbinned data
Y values
X values
72
64
56
48
40
32
24
16
8
0


Z = F(X, Y)
Unbinned data
Y values
X values
14
12
10
8
6
4
2


Z = F(X, Y)
Unbinned data
Y values
X values
Z = F(X, Y)
Unbinned data
Y values
X values
X, Y vs The No. of Pts Per Bin
Unbinned data
Y values
X values

Matplotlib: loading a colormap dynamically

In a thread on the matplotlib mailling list, James Boyle posted a way to load colormaps from a file. Here it is slightly modified.

gmtColormap.py

```
def gmtColormap(fileName,GMTPath = None):
      import colorsys
      import Numeric
      N = Numeric
      if type(GMTPath) == type(None):
          filePath = "/usr/local/cmaps/"+ fileName+".cpt"
      else:
          filePath = GMTPath+"/"+ fileName +".cpt"
      try:
          f = open(filePath)
      except:
          print "file ",filePath, "not found"
          return None

      lines = f.readlines()
      f.close()

      x = []
      r = []
      g = []
      b = []
      colorModel = "RGB"
      for l in lines:
          ls = l.split()
          if l[0] == "#":
             if ls[-1] == "HSV":
                 colorModel = "HSV"
                 continue
             else:
                 continue
          if ls[0] == "B" or ls[0] == "F" or ls[0] == "N":
             pass
          else:
              x.append(float(ls[0]))
              r.append(float(ls[1]))
              g.append(float(ls[2]))
              b.append(float(ls[3]))
              xtemp = float(ls[4])
              rtemp = float(ls[5])
              gtemp = float(ls[6])
              btemp = float(ls[7])
```

```
    x.append(xtemp)
    r.append(rtemp)
    g.append(gtemp)
    b.append(btemp)

    nTable = len(r)
    x = N.array( x , N.Float)
    r = N.array( r , N.Float)
    g = N.array( g , N.Float)
    b = N.array( b , N.Float)
    if colorModel == "HSV":
       for i in range(r.shape[0]):
           rr,gg,bb = colorsys.hsv_to_rgb(r[i]/360.,g[i],b[i])
           r[i] = rr ; g[i] = gg ; b[i] = bb
    if colorModel == "HSV":
       for i in range(r.shape[0]):
           rr,gg,bb = colorsys.hsv_to_rgb(r[i]/360.,g[i],b[i])
           r[i] = rr ; g[i] = gg ; b[i] = bb
    if colorModel == "RGB":
        r = r/255.
        g = g/255.
        b = b/255.
    xNorm = (x - x[0])/(x[-1] - x[0])

    red = []
    blue = []
    green = []
    for i in range(len(x)):
        red.append([xNorm[i],r[i],r[i]])
        green.append([xNorm[i],g[i],g[i]])
        blue.append([xNorm[i],b[i],b[i]])
    colorDict = {"red":red, "green":green, "blue":blue}
    return (colorDict)
```

Matplotlib: plotting images with special values

Image plotting requires data, a colormap, and a normalization. A common desire is to show missing data or other values in a specified color. The following code shows an example of how to do this.

The code creates a new Colormap subclass and a norm subclass.

The initialization takes a dictionary of value, color pairs. The data is already assumed to be normalized (except for the sentinels which are preserved). The RGB values at the sentinel values are replaced by the specified colors.

The class normalizes the data in the standard way except for one sublety. takes an “ignore” argument. The ignored values need to be excluded from the normalization so that they do not skew the results.

I use a not particularly wonderful algorithm of explicitly sorting the data and using the first non-sentinel values to define the min and max. This can probably be improved, but for my purposes was easy and sufficient. The data is then normalized including the sentinels. Finally, the sentinels are replaced.

```
from matplotlib.colors import Colormap, normalize
import matplotlib.numerix as nx
from types import IntType, FloatType, ListType

class SentinelMap(Colormap):
        def __init__(self, cmap, sentinels={}):
                # boilerplate stuff
                self.N = cmap.N
                self.name = 'SentinelMap'
                self.cmap = cmap
                self.sentinels = sentinels
                for rgb in sentinels.values():
                        if len(rgb)!=3:
                                raise ValueError('sentinel color mu

        def __call__(self, scaledImageData, alpha=1):
                # assumes the data is already normalized (ignoring
                # clip to be on the safe side
                rgbaValues = self.cmap(nx.clip(scaledImageData, 0.,

                #replace sentinel data with sentinel colors
                for sentinel,rgb in self.sentinels.items():
                        r,g,b = rgb
                        rgbaValues[:,:,0] =  nx.where(scaledImageDa
                        rgbaValues[:,:,1] =  nx.where(scaledImageDa
                        rgbaValues[:,:,2] =  nx.where(scaledImageDa
                        rgbaValues[:,:,3] =  nx.where(scaledImageDa
```

```
                return rgbaValues

class SentinelNorm(normalize):
        """
 Leave the sentinel unchanged
 """
        def __init__(self, ignore=[], vmin=None, vmax=None):
                self.vmin=vmin
                self.vmax=vmax

                if type(ignore) in [IntType, FloatType]:
                        self.ignore = [ignore]
                else:
                        self.ignore = list(ignore)

        def __call__(self, value):

                vmin = self.vmin
                vmax = self.vmax

                if type(value) in [IntType, FloatType]:
                        vtype = 'scalar'
                        val = array([value])
                else:
                        vtype = 'array'
                        val = nx.asarray(value)

                # if both vmin is None and vmax is None, we'll auto
                # norm the data to vmin/vmax of the actual data, so
                # clipping step won't be needed.
                if vmin is None and vmax is None:
                        needs_clipping = False
                else:
                        needs_clipping = True

                if vmin is None or vmax is None:
                        rval = nx.ravel(val)
                        #do this if sentinels (values to ignore in
                        if self.ignore:
                                sortValues=nx.sort(rval)
                                if vmin is None:
                                        # find the lowest non-senti
                                        for thisVal in sortValues:
                                                if thisVal not in s
                                                        vmin=thisVa
                                                        break
                                        else:
                                                vmin=0.
                                if vmax is None:
                                        for thisVal in sortValues[:
                                                if thisVal not in s
                                                        vmax=thisVa
                                                        break
```

```
                                        else:
                                                vmax=0.
                        else:
                                if vmin is None: vmin = min(rval)
                                if vmax is None: vmax = max(rval)
                if vmin > vmax:
                        raise ValueError("minvalue must be less tha
                elif vmin==vmax:
                        return 0.*value
                else:
                        if needs_clipping:
                                val = nx.clip(val,vmin, vmax)
                        result = (1.0/(vmax-vmin))*(val-vmin)

                # replace sentinels with original (non-normalized)
                for thisIgnore in self.ignore:
                        result = nx.where(val==thisIgnore,thisIgnor

                if vtype == 'scalar':
                        result = result[0]
                return result

if __name__=="__main__":
        import pylab
        import matplotlib.colors
        n=100

        # create a random array
        X = nx.mlab.rand(n,n)
        cmBase = pylab.cm.jet

        # plot it array as an image
        pylab.figure(1)
        pylab.imshow(X, cmap=cmBase, interpolation='nearest')

        # define the sentinels
        sentinel1 = -10
        sentinel2 = 10

        # replace some data with sentinels
        X[int(.1*n):int(.2*n), int(.5*n):int(.7*n)]  = sentinel1
        X[int(.6*n):int(.8*n), int(.2*n):int(.3*n)]  = sentinel2

        # define the colormap and norm
        rgb1 = (0.,0.,0.)
        rgb2 = (1.,0.,0.)
        cmap = SentinelMap(cmBase, sentinels={sentinel1:rgb1,sentin
        norm = SentinelNorm(ignore=[sentinel1,sentinel2])

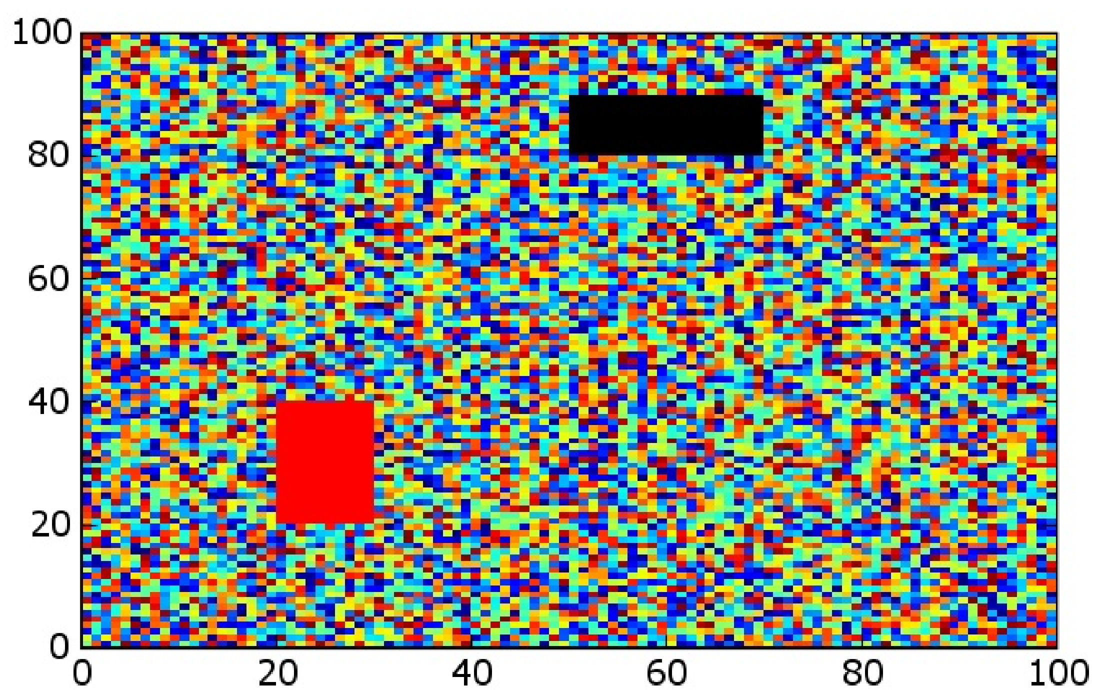
        # plot with the modified colormap and norm
        pylab.figure(2)
        pylab.imshow(X, cmap = cmap, norm=norm, interpolation='near
```

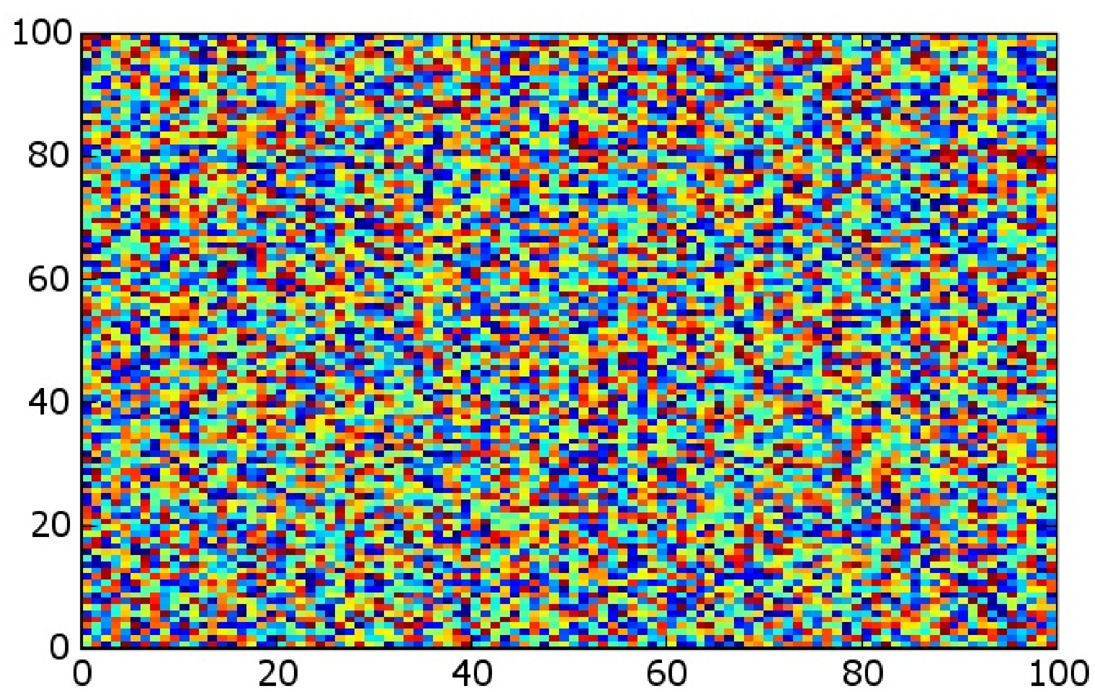
```
        pylab.show()
```

If the preceeding code is run from a prompt, two images are generated. The first is a pristine image of random data. The second image is the data modified by setting some blocks to sentinel values and then plotting the sentinels in specific colors. A sample result is shown below.

Attachments

- `sentinel.png`
- `sentinel_pristine.png`

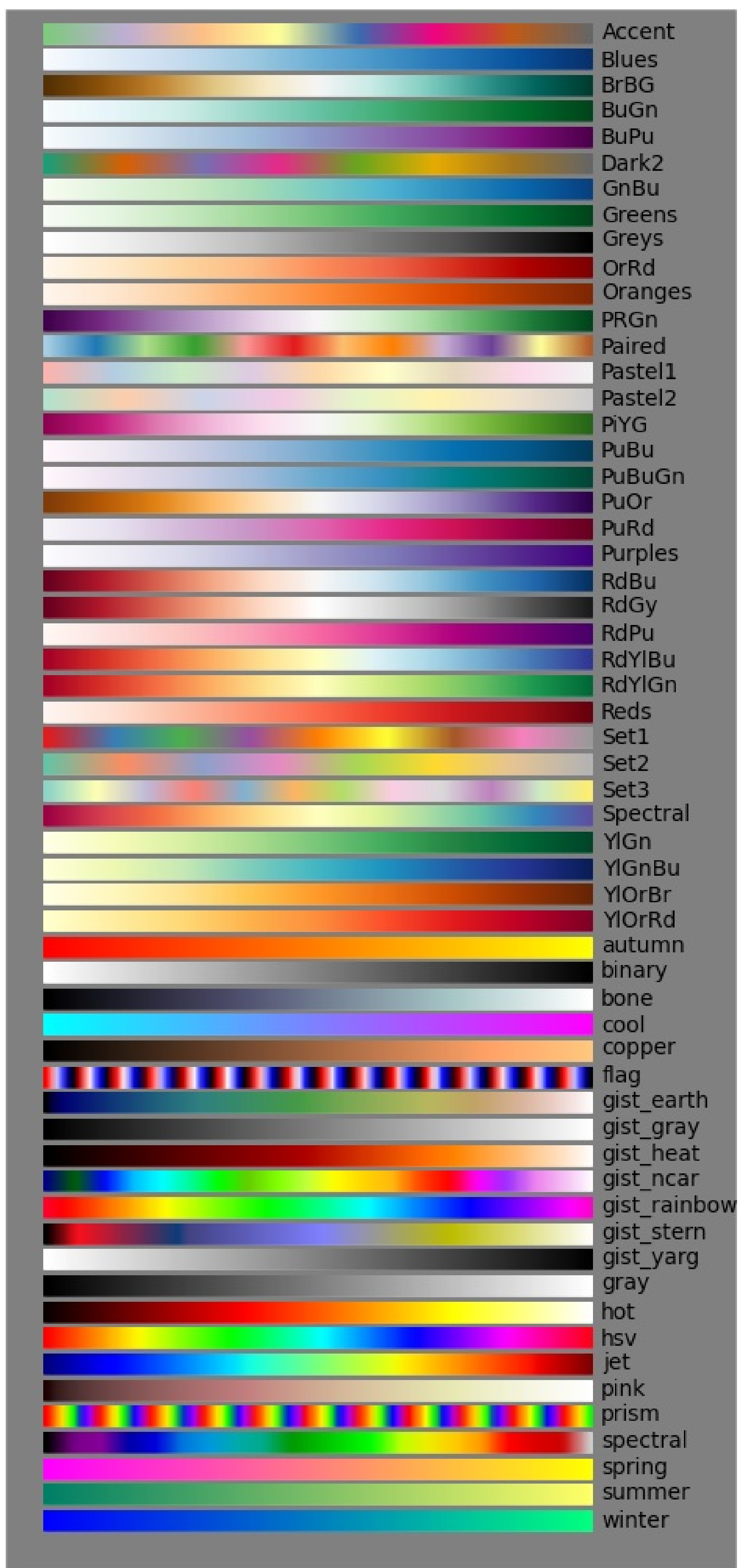

100
80
60
40
20
0
0
20
40
60
80
100


100
80
60
40
20
0
0
20
40
60
80
100

Matplotlib: show colormaps

Show Matplotlib colormaps

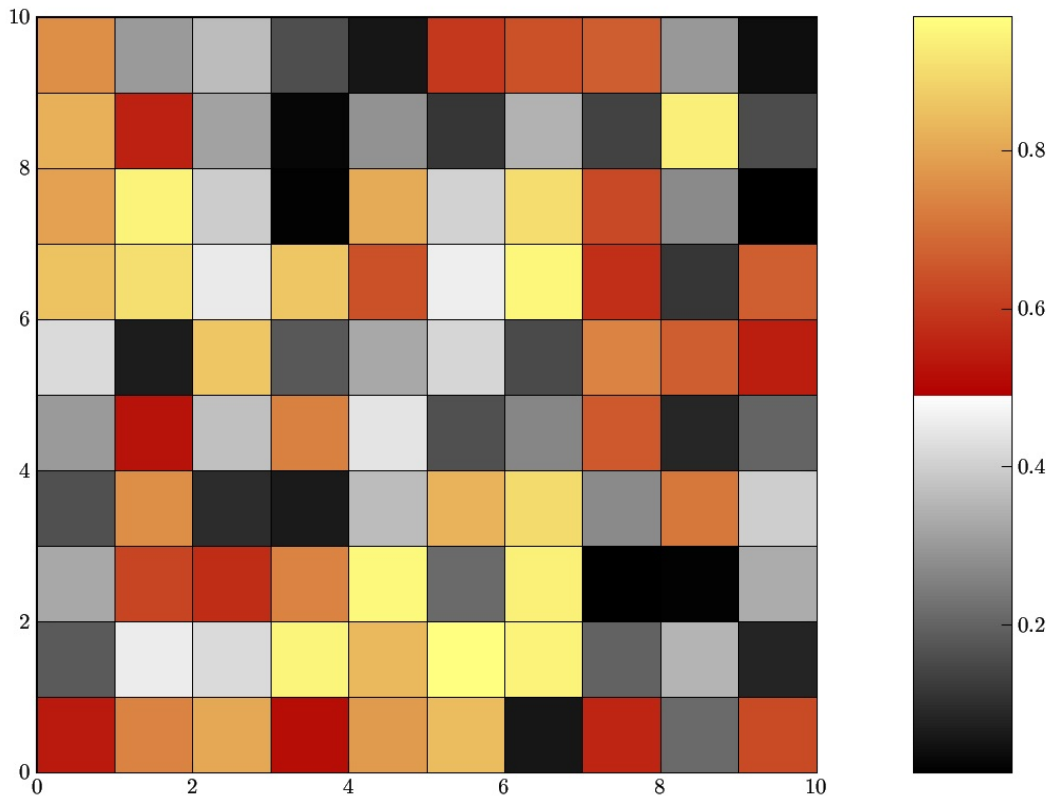
```
#!python
from pylab import *
from numpy import outer
rc('text', usetex=False)
a=outer(arange(0,1,0.01),ones(10))
figure(figsize=(10,5))
subplots_adjust(top=0.8,bottom=0.05,left=0.01,right=0.99)
maps=[m for m in cm.datad if not m.endswith("_r")]
maps.sort()
l=len(maps)+1
for i, m in enumerate(maps):
    subplot(1,l,i+1)
    axis("off")
    imshow(a,aspect='auto',cmap=get_cmap(m),origin="lower")
    title(m,rotation=90,fontsize=10)
savefig("colormaps.png",dpi=100,facecolor='gray')
```


Accent
Blues
BrBG
BuGn
BuPu
Dark2
GnBu
Greens
Greys
OrRd
Oranges
PRGn
Paired
Pastel1
Pastel2
PiYG
PuBu
PuBuGn
PuOr
PuRd
Purples
RdBu
RdGy
RdPu
RdYlBu
RdYlGn
Reds
Set1
Set2
Set3
Spectral
YlGn
YlGnBu
YlOrBr
YlOrRd
autumn
binary
bone
cool
copper
flag
gist_earth
gist_gray
gist_heat
gist_ncar
gist_rainbow
gist_stern
gist_yarg
gray
hot
hsv
jet
pink
prism
spectral
spring
summer
winter

But, what if I think those colormaps are ugly? Well, just make your own using matplotlib.colors.!LinearSegmentedColormap.

First, create a script that will map the range (0,1) to values in the RGB spectrum. In this dictionary, you will have a series of tuples for each color ‘red’, ‘green’, and ‘blue’. The first elements in each of these color series needs to be ordered from 0 to 1, with arbitrary spacing inbetween. Now, consider (0.5, 1.0, 0.7) in the ‘red’ series below. This tuple says that at 0.5 in the range from (0,1) , interpolate from below to 1.0, and above from 0.7. Often, the second two values in each tuple will be the same, but using diferent values is helpful for putting breaks in your colormap. This is easier understand than might sound, as demonstrated by this simple script:

```
#!python
from pylab import *
cdict = {'red': ((0.0, 0.0, 0.0),
                 (0.5, 1.0, 0.7),
                 (1.0, 1.0, 1.0)),
         'green': ((0.0, 0.0, 0.0),
                   (0.5, 1.0, 0.0),
                   (1.0, 1.0, 1.0)),
         'blue': ((0.0, 0.0, 0.0),
                  (0.5, 1.0, 0.0),
                  (1.0, 0.5, 1.0))}
my_cmap = matplotlib.colors.LinearSegmentedColormap('my_colormap',c
pcolor(rand(10,10),cmap=my_cmap)
colorbar()
```



As you see, the colormap has a break halfway through. Please use this new power responsibly.

Here a slightly modified version of the above code which allows for displaying a selection of the pre-defined colormaps as well as self-created registered colormaps. Note that the cmap_d dictionary in the cm module is not documented. The choice of indexed colors in discrete_cmap is somewhat haphazardous...

```
"""Python colormaps demo

includes:
examples for registering own color maps
utility for showing all or selected named colormaps including self-

import matplotlib
import matplotlib.colors as col
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import numpy as np

def register_own_cmaps():
    """define two example colormaps as segmented lists and register
    # a good guide for choosing colors is provided at
    # http://geography.uoregon.edu/datagraphics/color_scales.htm
    #
```

```
    # example 1:
    # create own colormap from purple, blue, green, orange to red
    # cdict contains a tuple structure for 'red', 'green', and 'blu
    # Each color has a list of (x,y0,y1) tuples, where
    # x defines the "index" in the colormap (range 0..1), y0 is the
    # color value (0..1) left of x, and y1 the color value right of
    # The LinearSegmentedColormap method will linearly interpolate
    # (x[i],y1) and (x[i+1],y0)
    # The gamma value denotes a "gamma curve" value which adjusts t
    # at the bottom and top of the colormap. According to matlab do
    # this means:
    # colormap values are modified as c^gamma, where gamma is (1-be
    # beta>0 and 1/(1+beta) for beta<=0
    cdict = {'red': ((0.0, 0.0, 0.0),
                     (0.3, 0.5, 0.5),
                     (0.6, 0.7, 0.7),
                     (0.9, 0.8, 0.8),
                     (1.0, 0.8, 0.8)),
         'green': ((0.0, 0.0, 0.0),
                   (0.3, 0.8, 0.8),
                   (0.6, 0.7, 0.7),
                   (0.9, 0.0, 0.0),
                   (1.0, 0.7, 0.7)),
         'blue': ((0.0, 1.0, 1.0),
                  (0.3, 1.0, 1.0),
                  (0.6, 0.0, 0.0),
                  (0.9, 0.0, 0.0),
                  (1.0, 1.0, 1.0))}
    cmap1 = col.LinearSegmentedColormap('my_colormap',cdict,N=256,g
    cm.register_cmap(name='own1', cmap=cmap1)

    # example 2: use the "fromList() method
    startcolor = '#586323'  # a dark olive
    midcolor = '#fcffc9'    # a bright yellow
    endcolor = '#bd2309'    # medium dark red
    cmap2 = col.LinearSegmentedColormap.from_list('own2',[startcolo
    # extra arguments are N=256, gamma=1.0
    cm.register_cmap(cmap=cmap2)
    # we can skip name here as it was already defined

def discrete_cmap(N=8):
    """create a colormap with N (N<15) discrete colors and registe
    # define individual colors as hex values
    cpool = [ '#bd2309', '#bbb12d', '#1480fa', '#14fa2f', '#000000
              '#faf214', '#2edfea', '#ea2ec4', '#ea2e40', '#cdcdcd
              '#577a4d', '#2e46c0', '#f59422', '#219774', '#8086d9
    cmap3 = col.ListedColormap(cpool[0:N], 'indexed')
    cm.register_cmap(cmap=cmap3)

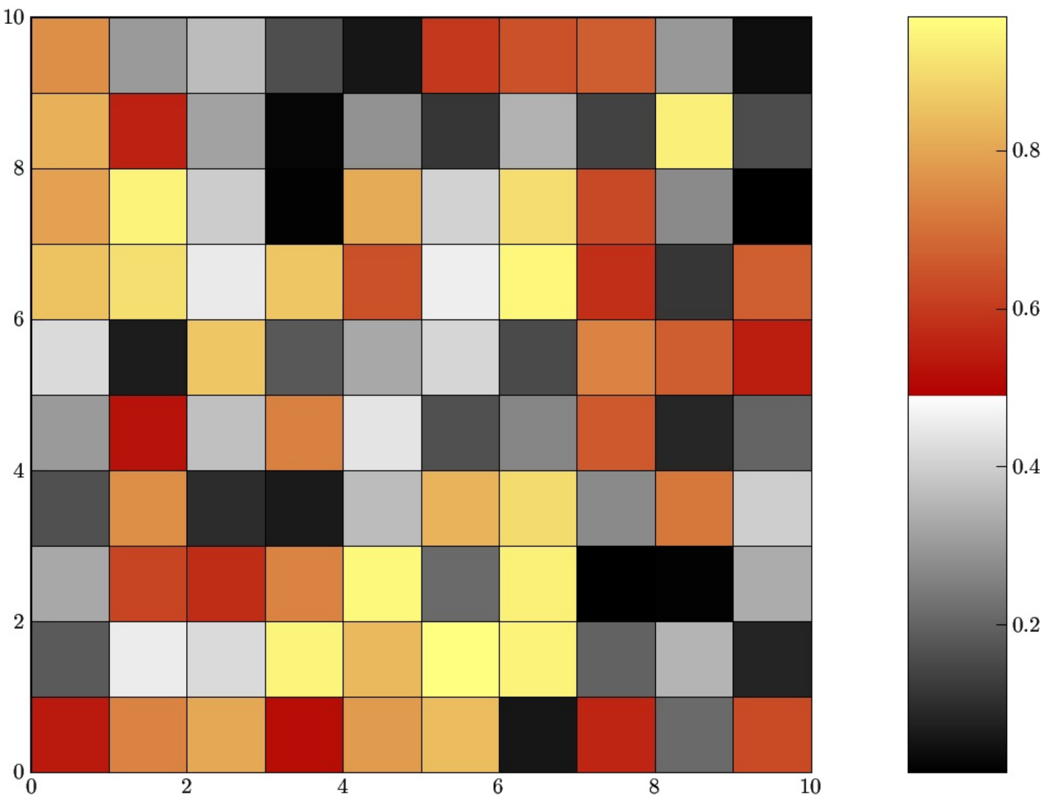
def show_cmaps(names=None):
    """display all colormaps included in the names list. If names i
defined colormaps will be shown."""
    # base code from http://www.scipy.org/Cookbook/Matplotlib/Show_
```

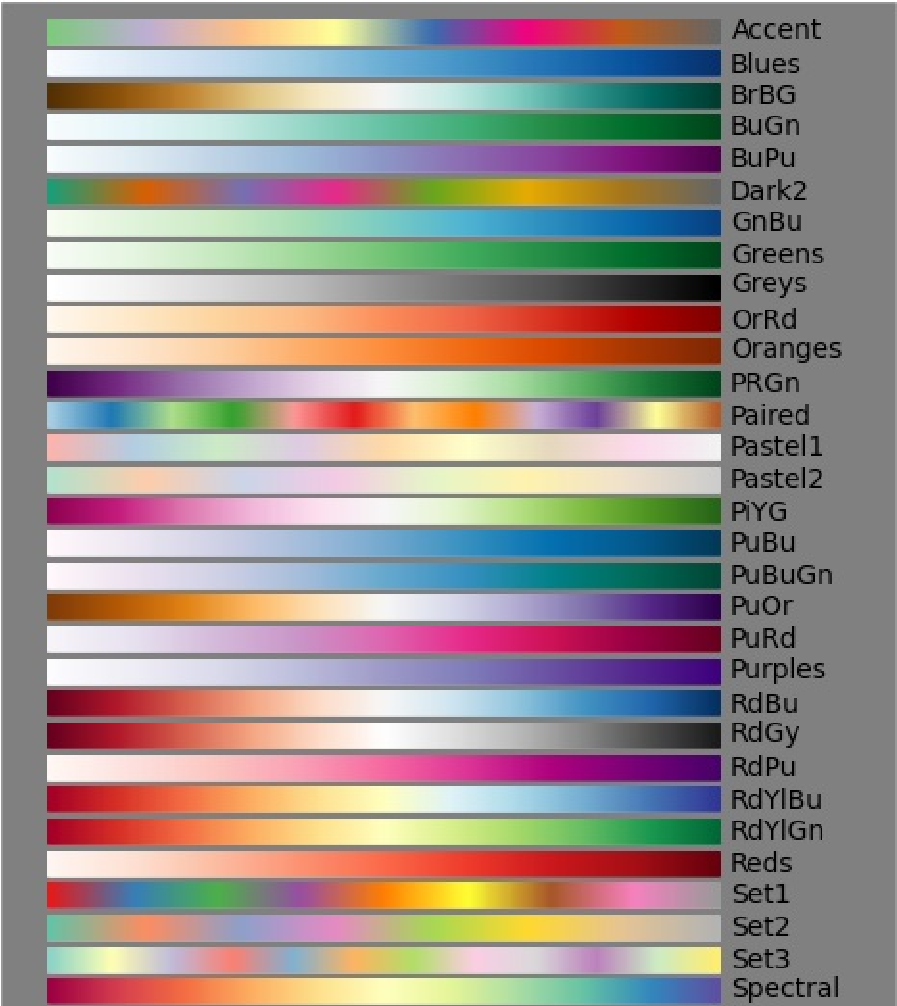
```
    matplotlib.rc('text', usetex=False)
    a=np.outer(np.arange(0,1,0.01),np.ones(10))   # pseudo image da
    f=plt.figure(figsize=(10,5))
    f.subplots_adjust(top=0.8,bottom=0.05,left=0.01,right=0.99)
    # get list of all colormap names
    # this only obtains names of built-in colormaps:
    maps=[m for m in cm.datad if not m.endswith("_r")]
    # use undocumented cmap_d dictionary instead
    maps = [m for m in cm.cmap_d if not m.endswith("_r")]
    maps.sort()
    # determine number of subplots to make
    l=len(maps)+1
    if names is not None: l=len(names)  # assume all names are corr
    # loop over maps and plot the selected ones
    i=0
    for m in maps:
        if names is None or m in names:
            i+=1
            ax = plt.subplot(1,l,i)
            ax.axis("off")
            plt.imshow(a,aspect='auto',cmap=cm.get_cmap(m),origin='
            plt.title(m,rotation=90,fontsize=10,verticalalignment=
    plt.savefig("colormaps.png",dpi=100,facecolor='gray')

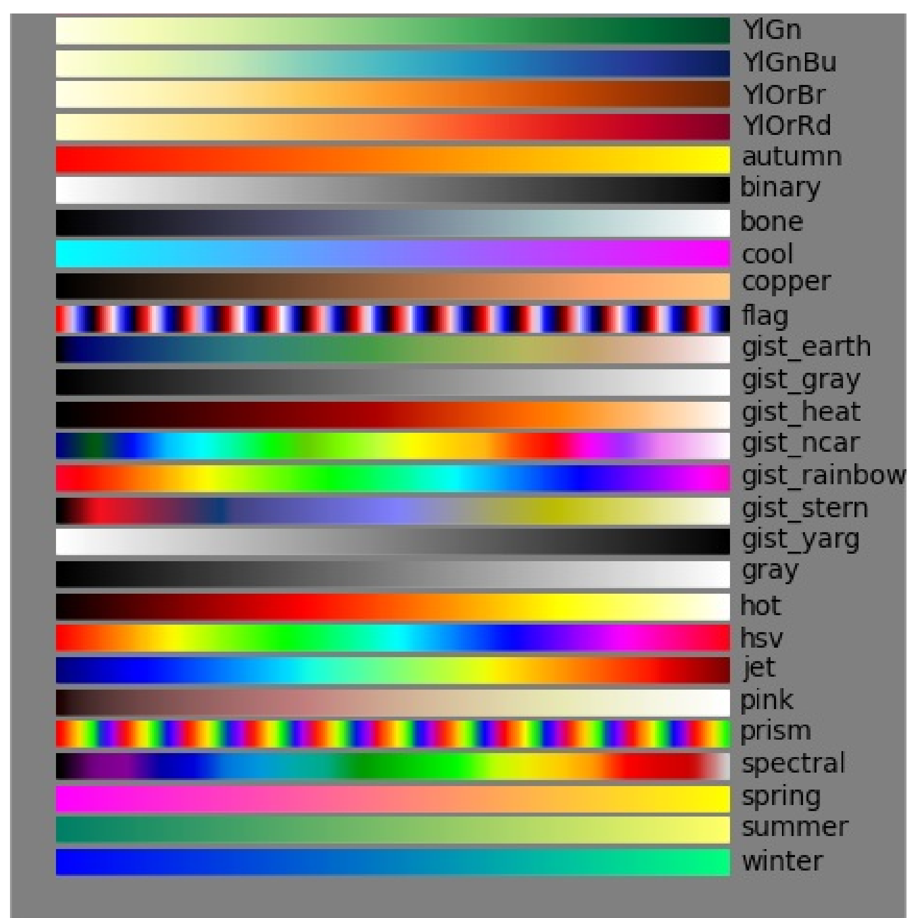
if __name__ == "__main__":
    register_own_cmaps()
    discrete_cmap(8)
    show_cmaps(['indexed','Blues','OrRd','PiYG','PuOr',
                'RdYlBu','RdYlGn','afmhot','binary','copper',
                'gist_ncar','gist_rainbow','own1','own2'])
```

Attachments

- `cmap_example.png`
- `colormaps3.png`


10
8
6
4
2
0
0
2
4
6
8
10
0.8
0.6
0.4
0.2


Accent
Blues
BrBG
BuGn
BuPu
Dark2
GnBu
Greens
Greys
OrRd
Oranges
PRGn
Paired
Pastel1
Pastel2
PiYG
PuBu
PuBuGn
PuOr
PuRd
Purples
RdBu
RdGy
RdPu
RdYlBu
RdYlGn
Reds
Set1
Set2
Set3
Spectral


YlGn
YlGnBu
YlOrBr
YlOrRd
autumn
binary
bone
cool
copper
flag
gist_earth
gist_gray
gist_heat
gist_ncar
gist_rainbow
gist_stern
gist_yarg
gray
hot
hsv
jet
pink
prism
spectral
spring
summer
winter

Matplotlib / Simple Plotting

Matplotlib: animations

Note: Some of the matplotlib code in this cookbook entry may be deprecated or obsolete. For example, the file **anim.py** mentioned below no longer exists in matplotlib. Examples of animation in matplotlib are at http://matplotlib.sourceforge.net/examples/animation/index.html, and the main animation API documentation is http://matplotlib.sourceforge.net/api/animation_api.html.

---

matplotlib supports animated plots, and provides a number of demos. An important question when considering whether to use matplotlib for animation is what kind of speed you need. matplotlib is not the fastest plotting library in the west, and may be too slow for some animation applications. Nonetheless, it is fast enough for many if not most, and this tutorial is geared to showing you how to make it fast enough for you. In particular, the section *Animating selected plot elements* shows you how to get top speed out of matplotlib animations.

Performance

matplotlib supports 5 different graphical user interfaces (GTK, WX, Qt, Tkinter, FLTK) and for some of those GUIs, there are various ways to draw to the canvas. For example, for GTK, you can use native GDK drawing, antigrain, or cairo. A GUI toolkit combined with some method of drawing comprises a backend. For example, drawing to a GTK canvas with the antigrain drawing toolkit is called the GTKAgg backend. This is important because different backends have different performance characteristics, and the difference can be considerable.

When considering performance, the typical measure is frames per second. Television is 30 frames per second, and for many application if you can get 10 or more frames per second the animation is smooth enough to “look good”. Monitors refresh at 75-80 frames per second typically, and so this is an upper limit for performance. Any faster is probably wasted CPU cycles.

Here are some numbers for the animated script anim.py, which simply updates a sine wave on various backends, run under linux on a 3GHz Pentium IV. To profile a script under different backends, you can use the “GUI neutral” animation technique described below and then run it with the flag, as in:

```
> python anim.py -dWX
> python anim.py -dGTKAgg
```

Here are the results. Note that these should be interpreted cautiously, because some GUIs might call a drawing operation in a separate thread and return before it is complete, or drop a drawing operation while one is in the queue. The most important assessment is qualitative.

```
Backend  Frames/second
 GTK         43
 GTKAgg      36
 TkAgg       20
 WX          11
 WXAgg       27
```

GUI neutral animation in pylab

The pylab interface supports animation that does not depend on a specific GUI toolkit. This is not recommended for production use, but is often a good way to make a quick-and-dirty, throw away animation. After importing pylab, you need to turn interaction on with the [http://matplotlib.sf.net/matplotlib.pylab.html#-ion ion] command. You can then force a draw at any time with [http://matplotlib.sf.net/matplotlib.pylab.html#-draw draw]. In interactive mode, a new draw command is issued after each pylab command, and you can also temporarily turn off this behavior for a block of plotting commands in which you do not want an update with the [http://matplotlib.sf.net/matplotlib.pylab.html#-ioff ioff] commands. This is described in more detail on the interactive page.

Here is the anim.py script that was used to generate the profiling numbers across backends in the table above

```
from pylab import *
import time

ion()

tstart = time.time()               # for profiling
x = arange(0,2*pi,0.01)            # x-array
line, = plot(x,sin(x))
for i in arange(1,200):
    line.set_ydata(sin(x+i/10.0))  # update the data
    draw()                         # redraw the canvas

print 'FPS:' , 200/(time.time()-tstart)
```

Note the technique of creating a line with the call to [http://matplotlib.sf.net/matplotlib.pylab.html#-plot plot]:

```
line, = plot(x,sin(x))
```

and then setting its data with the set_ydata method and calling draw:

```
line.set_ydata(sin(x+i/10.0))  # update the data
draw()                         # redraw the canvas
```

This can be much faster than clearing the axes and/or creating new objects for each plot command.

To animate a pcolor plot use:

```
p = pcolor(XI,YI,C[0])

for i in range(1,len(C)):
  p.set_array(C[i,0:-1,0:-1].ravel())
  p.autoscale()
  draw()
```

This assumes C is a 3D array where the first dimension is time and that XI,YI and C[i,:,:] have the same shape. If C[i,:,:] is one row and one column smaller simply use C.ravel().

Using the GUI timers or idle handlers

If you are doing production code or anything semi-serious, you are advised to use the GUI event handling specific to your toolkit for animation, because this will give you more control over your animation than matplotlib can provide through the GUI neutral pylab interface to animation. How you do this depends on your toolkit, but there are examples for several of the backends in the matplotlib examples directory, eg, anim_tk.py for Tkinter, dynamic_image_gtkagg.py for GTKAgg and dynamic_image_wxagg.py for WXAgg.

The basic idea is to create your figure and a callback function that updates your figure. You then pass that callback to the GUI idle handler or timer. A simple example in GTK looks like

```
def callback(*args):
   line.set_ydata(get_next_plot())
   canvas.draw()  # or simply "draw" in pylab

gtk.idle_add(callback)
```

A simple example in WX or WXAgg looks like

```
def callback(*args):
   line.set_ydata(get_next_plot())
   canvas.draw()
   wx.WakeUpIdle() # ensure that the idle event keeps firing

wx.EVT_IDLE(wx.GetApp(), callback)
```

Animating selected plot elements

One limitation of the methods presented above is that all figure elements are redrawn with every call to draw, but we are only updating a single element. Often what we want to do is draw a background, and animate just one or two elements on top of it. As of matplotlib-0.87, GTKAgg, !TkAgg, WXAgg, and FLTKAgg support the methods discussed here.

The basic idea is to set the ‘animated’ property of the Artist you want to animate (all figure elements from Figure to Axes to Line2D to Text derive from the base class Artist). Then, when the standard canvas draw operation is called, all the artists except the animated one will be drawn. You can then use the method to copy a rectangular region (eg the axes bounding box) into a a pixel buffer. In animation, you restore the background with , and then call to draw your animated artist onto the clean background, and to blit the updated axes rectangle to the figure. When I run the example below in the same environment that produced 36 FPS for GTKAgg above, I measure 327 FPS with the techniques below. See the caveats on performance numbers mentioned above. Suffice it to say, quantitatively and qualitiatively it is much faster.

```
import sys
import gtk, gobject
import pylab as p
import matplotlib.numerix as nx
import time

ax = p.subplot(111)
canvas = ax.figure.canvas

# for profiling
tstart = time.time()

# create the initial line
x = nx.arange(0,2*nx.pi,0.01)
line, = p.plot(x, nx.sin(x), animated=True)

# save the clean slate background -- everything but the animated li
# is drawn and saved in the pixel buffer background
background = canvas.copy_from_bbox(ax.bbox)

def update_line(*args):
    # restore the clean slate background
    canvas.restore_region(background)
    # update the data
    line.set_ydata(nx.sin(x+update_line.cnt/10.0))
    # just draw the animated artist
    ax.draw_artist(line)
    # just redraw the axes rectangle
    canvas.blit(ax.bbox)

    if update_line.cnt==50:
        # print the timing info and quit
        print 'FPS:' , update_line.cnt/(time.time()-tstart)
        sys.exit()

    update_line.cnt += 1
    return True
update_line.cnt = 0

gobject.idle_add(update_line)
p.show()
```

Example: cursoring

matplotlib 0.83.2 introduced a cursor class which can utilize these blit methods for no lag cursoring. The class takes a argument in the constructor. For backends that support the new API (GTKAgg) set :

```
from matplotlib.widgets import Cursor
import pylab

fig = pylab.figure(figsize=(8,6))
ax = fig.add_axes([0.075, 0.25, 0.9, 0.725], axisbg='#FFFFCC')

x,y = 4*(pylab.rand(2,100)-.5)
ax.plot(x,y,'o')
ax.set_xlim(-2,2)
ax.set_ylim(-2,2)

# set useblit = True on gtkagg for enhanced performance
cursor = Cursor(ax, useblit=True, color='red', linewidth=2 )

pylab.show()
```

The ‘blit’ animation methods

As noted above, only the GTKAgg supports the methods above to to the animations of selected actors. The following are needed

Figure canvas methods

```
* `` - copy the region in ax.bbox into a pixel buffer and return it
```

`module` <http://matplotlib.sf.net/transforms.html> __ .

```
is not used by the matplotlib frontend, but it stores it and passes
```

method. You will probably want to store not only the pixel buffer but the rectangular region of the canvas from whence it came in the background object.`

```
* `` - restore the region copied above to the canvas.
```

```
* `` - transfer the pixel buffer in region bounded by bbox to the ca
```

For *Agg backends, there is no need to implement the first two as Agg will do all the work (defines them). Thus you only need to be able to blit the agg buffer from a selected rectangle. One thing that might make this easier for backends using string methods to transfer the agg pixel buffer to their respective canvas is to define a method in agg. If you are working on this and need help, please contact the matplotlib-devel list.

Once all/most of the backends have implemented these methods, the matplotlib front end can do all the work of managing the background/restore/blit opertations, and userland animated code can look like

```
line, = plot(something, animated=True)
draw()
def callback(*args):
    line.set_ydata(somedata)
    ax.draw_animated()
```

and the rest will happen automagically. Since some backends ‘’‘do not’‘’ currently implement the required methods, I am making them available to the users to manage themselves but am not assuming them in the axes drawing code.

Matplotlib: arrows

Some example code for how to plot an arrow using the Arrow function.

```
from matplotlib.pyplot import *
from numpy import *

x = arange(10)
y = x

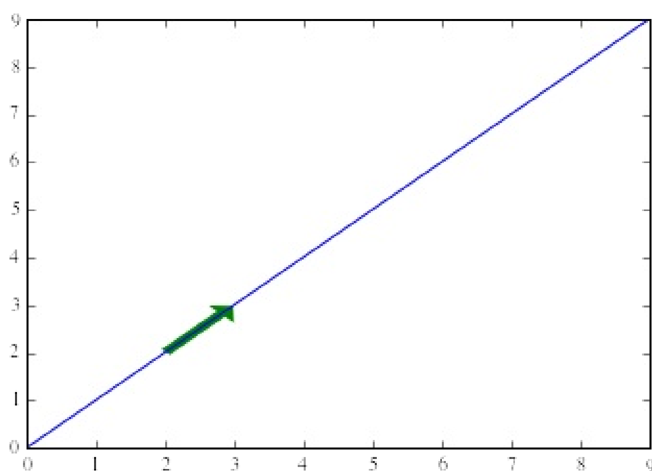
# Plot junk and then a filled region
plot(x, y)

# Now lets make an arrow object
arr = Arrow(2, 2, 1, 1, edgecolor='white')

# Get the subplot that we are currently working on
ax = gca()

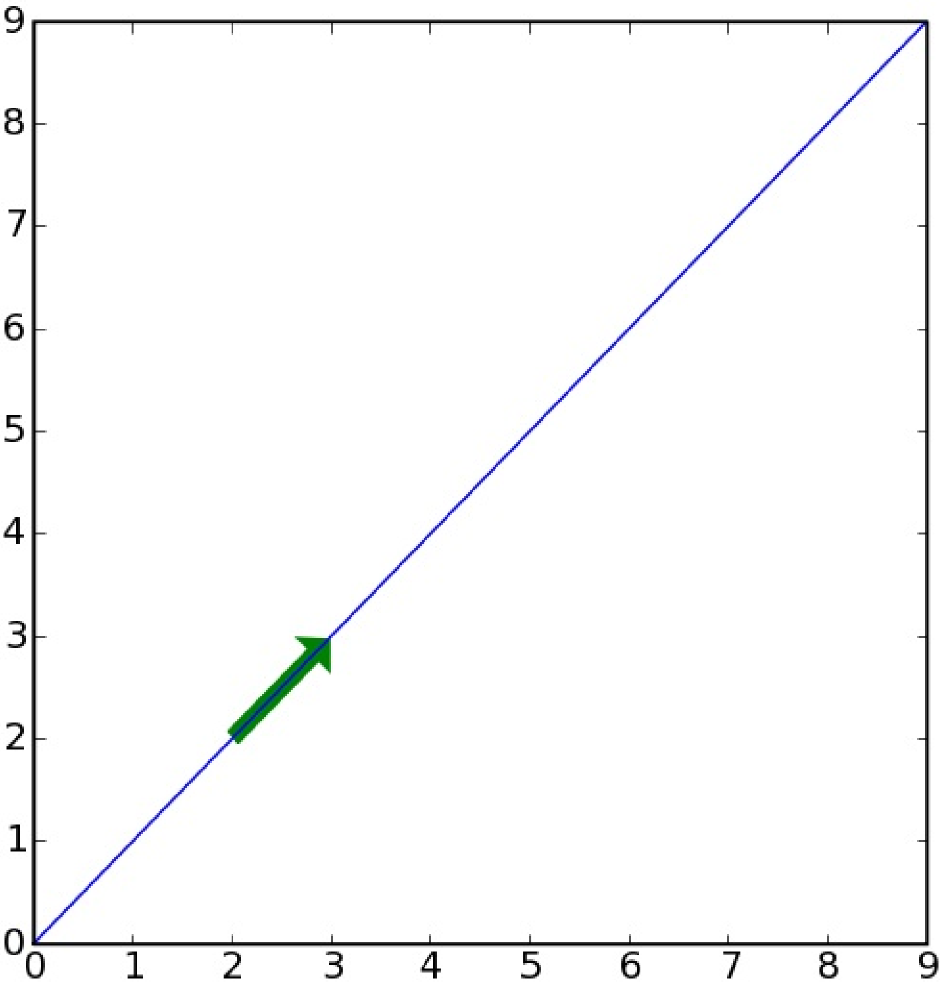
# Now add the arrow
ax.add_patch(arr)

# We should be able to make modifications to the arrow.
# Lets make it green.
arr.set_facecolor('g')
```



Attachments

- `plot_arrow.png`


9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9

Matplotlib: bar charts

Use the bar function to make bar charts:
http://matplotlib.sourceforge.net/matplotlib.pylab.html#-bar

Here's an example script that makes a bar char with error bars and labels centered under the bars.

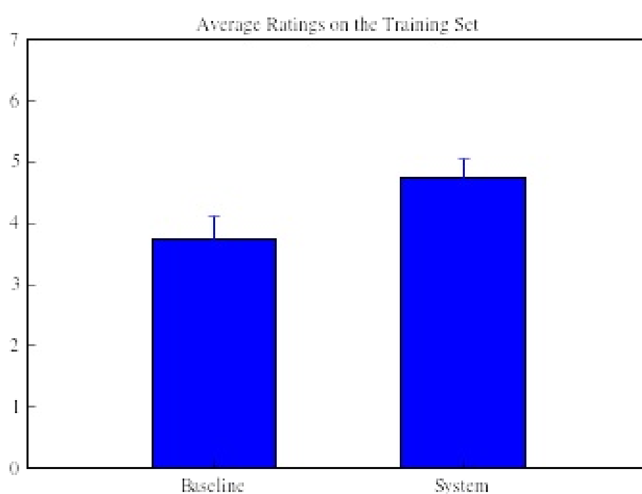
```
#!/usr/bin/env python
import numpy as na

from matplotlib.pyplot import *

labels = ["Baseline", "System"]
data =   [3.75               , 4.75]
error =  [0.3497             , 0.3108]

xlocations = na.array(range(len(data)))+0.5
width = 0.5
bar(xlocations, data, yerr=error, width=width)
yticks(range(0, 8))
xticks(xlocations+ width/2, labels)
xlim(0, xlocations[-1]+width*2)
title("Average Ratings on the Training Set")
gca().get_xaxis().tick_bottom()
gca().get_yaxis().tick_left()

show()
```



Attachments

- `barchart.png`


Average Ratings on the Training Set
7
6
5
4
3
2
1
0
Baseline
System

Matplotlib: custom log labels

Example of how to replace the default log-plot exponential labels with integer labels. The same method will work for any kind of custom labeling. This example was pulled from the Python-list mailing list and the original can be found here.

```
from matplotlib.pyplot import *

def log_10_product(x, pos):
    """The two args are the value and tick position.
 Label ticks with the product of the exponentiation"""
    return '%1i' % (x)

# Axis scale must be set prior to declaring the Formatter
# If it is not the Formatter will use the default log labels for t
ax = subplot(111)
ax.set_xscale('log')
ax.set_yscale('log')

formatter = FuncFormatter(log_10_product)
ax.xaxis.set_major_formatter(formatter)
ax.yaxis.set_major_formatter(formatter)

# Create some artificial data.
result1 = [3, 5, 70, 700, 900]
result2 = [1000, 2000, 3000, 4000, 5000]
predict1 = [4, 8, 120, 160, 200]
predict2 = [2000, 4000, 6000, 8000, 1000]

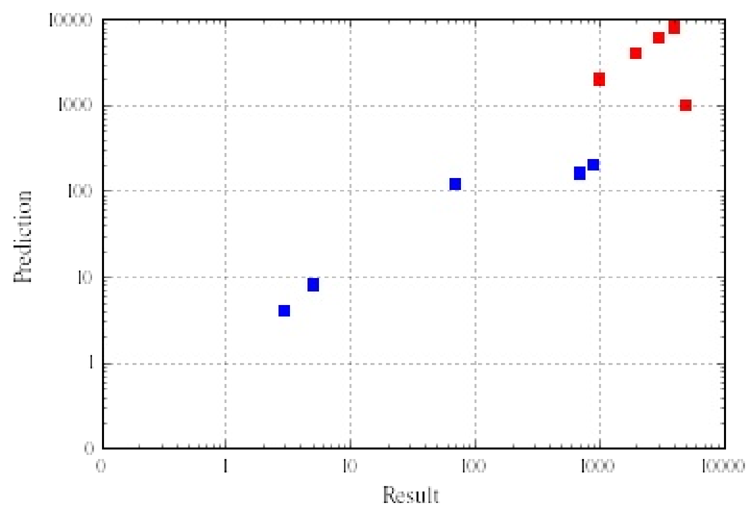
# Plot
ax.scatter(result1, predict1, s=40, c='b', marker='s', faceted=Fals
ax.scatter(result2, predict2, s=40, c='r', marker='s', faceted=Fals

ax.set_xlim(1e-1, 1e4)
ax.set_ylim(1e-1, 1e4)
grid(True)

xlabel(r"Result", fontsize = 12)
ylabel(r"Prediction", fontsize = 12)
```

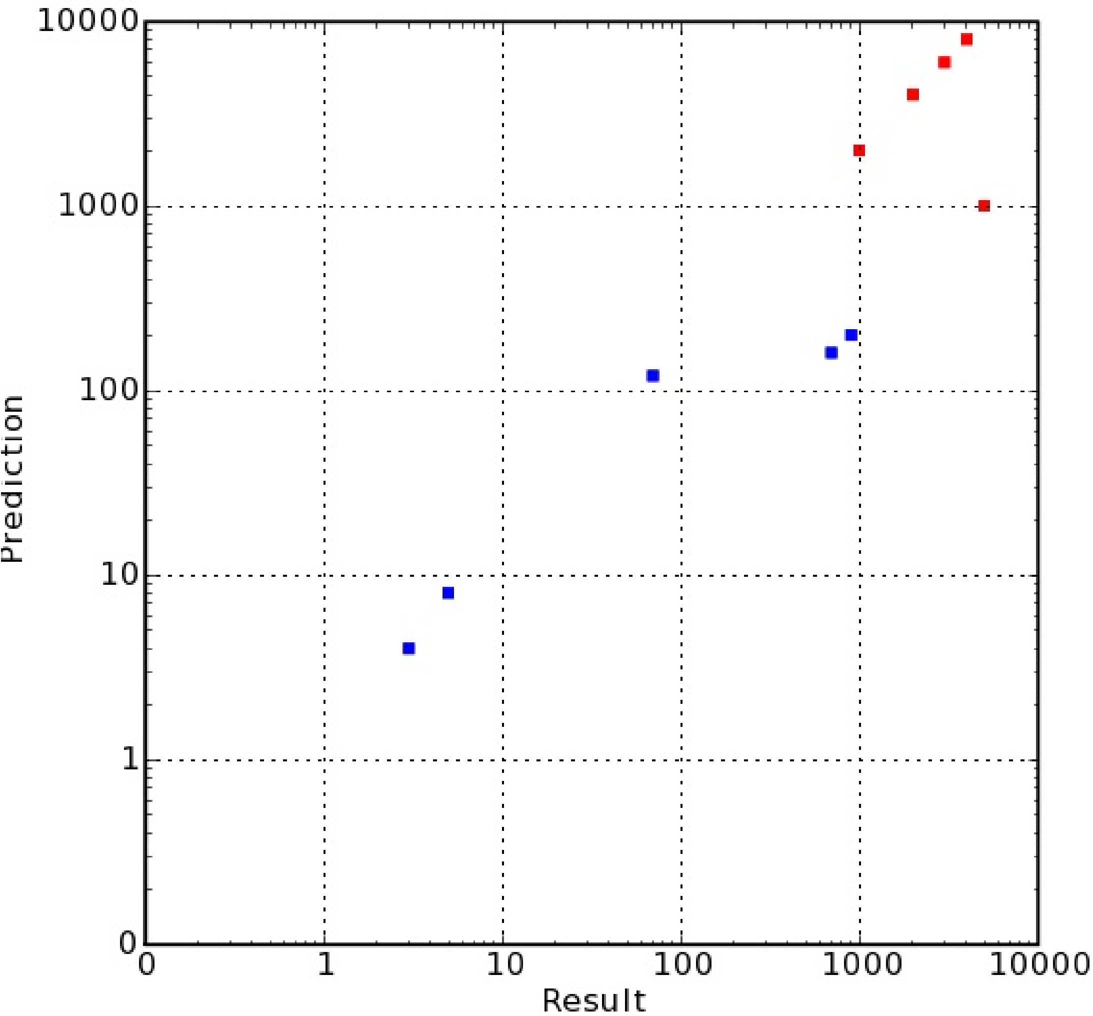
```
 /usr/lib/python2.7/dist-packages/matplotlib/cbook.py:137: Matplot
   warnings.warn(message, mplDeprecation, stacklevel=1)
```

```
<matplotlib.text.Text at 0x7f07874f2410>
```



Attachments

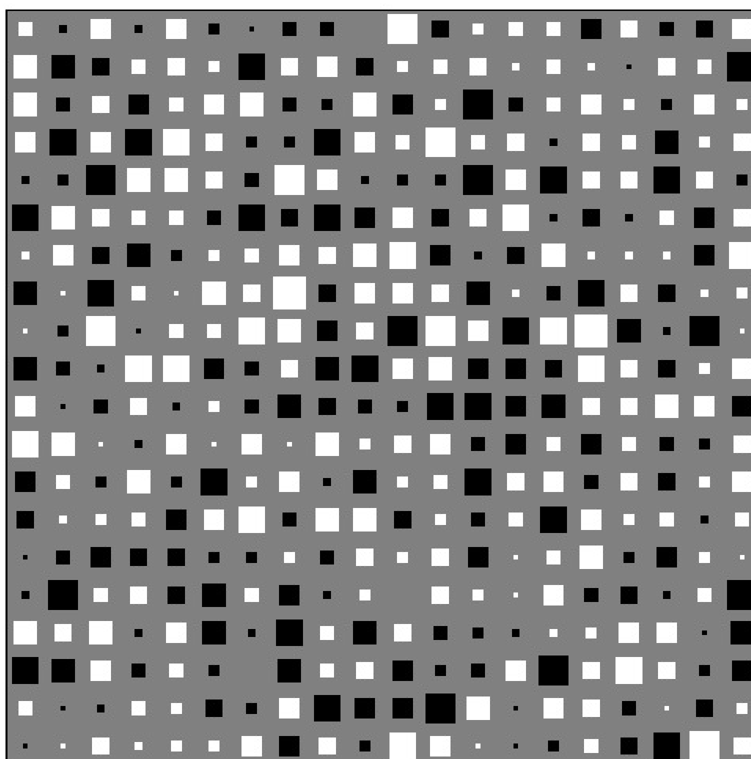
- `log_labels.png`


10000
1000
100
10
1
0
Prediction
0
1
10
100
1000
10000
Result

Matplotlib: hint on diagrams

Hinton diagrams with matplotlib

Hinton diagrams are a way of visualizing numerical values in a matrix/vector, popular in the neural networks and machine learning literature. The area occupied by a square is proportional to a value's magnitude, and the colour (black or white in this case) indicates its sign (positive/negative).



)#

```
import numpy as N
import pylab as P

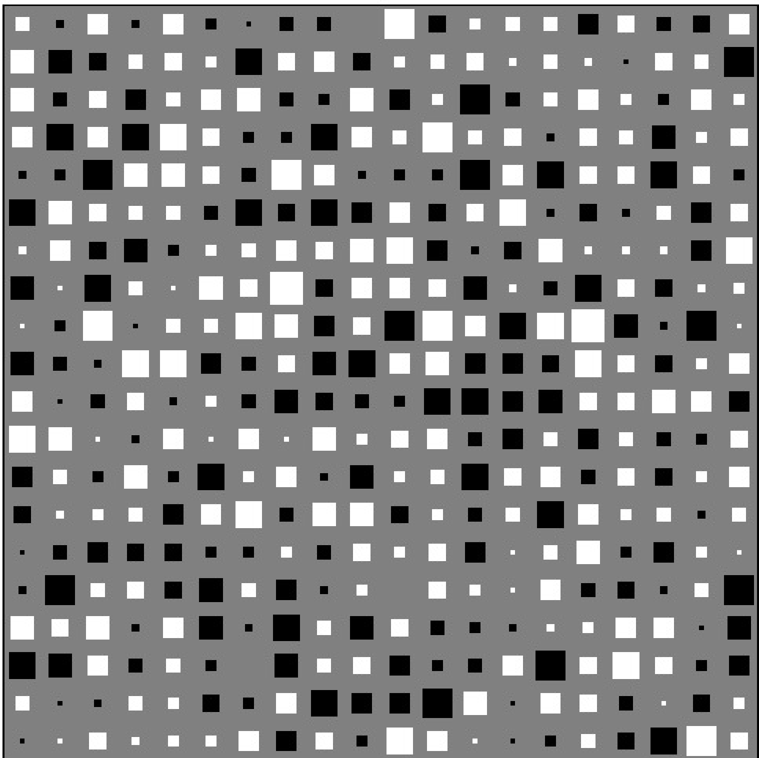
def _blob(x,y,area,colour):
    """
 Draws a square-shaped blob with the given area (< 1) at
 the given coordinates.
 """
    hs = N.sqrt(area) / 2
    xcorners = N.array([x - hs, x + hs, x + hs, x - hs])
    ycorners = N.array([y - hs, y - hs, y + hs, y + hs])
    P.fill(xcorners, ycorners, colour, edgecolor=colour)

def hinton(W, maxWeight=None):
    """
 Draws a Hinton diagram for visualizing a weight matrix.
 Temporarily disables matplotlib interactive mode if it is on,
 otherwise this takes forever.
 """
    reenable = False
    if P.isinteractive():
        P.ioff()
    P.clf()
    height, width = W.shape
    if not maxWeight:
        maxWeight = 2**N.ceil(N.log(N.max(N.abs(W)))/N.log(2))

    P.fill(N.array([0,width,width,0]),N.array([0,0,height,height]),
    P.axis('off')
    P.axis('equal')
    for x in xrange(width):
        for y in xrange(height):
            _x = x+1
            _y = y+1
            w = W[y,x]
            if w > 0:
                _blob(_x - 0.5, height - _y + 0.5, min(1,w/maxWeigh
            elif w < 0:
                _blob(_x - 0.5, height - _y + 0.5, min(1,-w/maxWeig
    if reenable:
        P.ion()
    P.show()
```

Attachments

- `hinton.png`



Matplotlib: legend

Legends for overlaid lines and markers

If you have a lot of points to plot, you may want to set a marker only once every *n* points, e.g.

```
plot(x, y, '-r')
plot(x[::20], y[::20], 'ro')
```

Then the automatic legend sees this as two different plots. One approach is to create an extra line object that is not plotted anywhere but used only for the legend:

```
from matplotlib.lines import Line2D
line = Line2D(range(10), range(10), linestyle='-', marker='o')
legend((line,), (label,))
```

Another possibility is to modify the line object within the legend:

```
line = plot(x, y, '-r')
markers = plot(x[::20], y[::20], 'ro')
lgd = legend([line], ['data'], numpoints=3)
lgd.get_lines()[0].set_marker('o')
draw()
```

Legend outside plot

There is no nice easy way to add a legend outside (to the right of) your plot, but if you set the axes right in the first place, it works OK:

```
figure()
axes([0.1,0.1,0.71,0.8])
plot([0,1],[0,1],label="line 1")
hold(1)
plot([0,1],[1,0.5],label="line 2")
legend(loc=(1.03,0.2))
show()
```

Removing a legend from a plot

One can simply set the attribute of the axes to and redraw:

```
ax = gca()
ax.legend_ = None
draw()
```

If you find yourself doing this often use a function such as

```
def remove_legend(ax=None):
    """Remove legend for ax or the current axes."""

    from pylab import gca, draw
    if ax is None:
        ax = gca()
    ax.legend_ = None
    draw()
```

(Source: Re: How to delete legend with matplotlib OO from a graph?)

Changing the font size of legend text

Note that to set the default font size, one can change the *legend.size* property in the matplotlib rc parameters file.

To change the font size for just one plot, use the matplotlib.font*manager.FontProperties argument, _prop*, for the legend creation.

```
x = range(10)
y = range(10)
handles = plot(x, y)
legend(handles, ["label1"], prop={"size":12})
```

Matplotlib: maps

Plotting data on map projections is easy with the basemap toolkit. Toolkits are collections of application-specific functions that extend matplotlib.

The basemap toolkit is not in the default matplotlib install - you can download it from the matplotlib sourceforge download page.

Suppose you'd like to make a map of the world using an orthographic, or satellite projection and plot some data on it. Here's how to make the map (using matplotlib >= 0.98.0 and basemap >= 0.99):

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np
# set up orthographic map projection with
# perspective of satellite looking down at 50N, 100W.
# use low resolution coastlines.
# don't plot features that are smaller than 1000 square km.
map = Basemap(projection='ortho', lat_0 = 50, lon_0 = -100,
              resolution = 'l', area_thresh = 1000.)
# draw coastlines, country boundaries, fill continents.
map.drawcoastlines()
map.drawcountries()
map.fillcontinents(color = 'coral')
# draw the edge of the map projection region (the projection limb)
map.drawmapboundary()
# draw lat/lon grid lines every 30 degrees.
map.drawmeridians(np.arange(0, 360, 30))
map.drawparallels(np.arange(-90, 90, 30))
plt.show()
```

There are many other map projections available, probably more than you've even heard of before. A complete list is available in the basemap docstrings. Coastlines, political boundaries and rivers are available in four resolutions, , , and . Here's what the resolution coastlines look like.

[](files/attachments/Matplotlib_Maps/basemap0.png

Now, suppose you would like to plot the locations of five cities on this map. Add the following just before the in the above script:

```
# lat/lon coordinates of five cities.
lats = [40.02, 32.73, 38.55, 48.25, 17.29]
lons = [-105.16, -117.16, -77.00, -114.21, -88.10]
cities=['Boulder, CO','San Diego, CA',
        'Washington, DC','Whitefish, MT','Belize City, Belize']
# compute the native map projection coordinates for cities.
x,y = map(lons,lats)
# plot filled circles at the locations of the cities.
map.plot(x,y,'bo')
# plot the names of those five cities.
for name,xpt,ypt in zip(cities,x,y):
    plt.text(xpt+50000,ypt+50000,name)
```

[](files/attachments/Matplotlib_Maps/basemap1b.png

Calling a basemap class instance with arrays of longitudes and latitudes returns those locations in native map projection coordinates using the proj4 library. Now suppose you have some data on a regular latitude/longitude grid and you would like to plot contours of that data over the map. Try adding the following lines just before

```
# make up some data on a regular lat/lon grid.
nlats = 73; nlons = 145; delta = 2.*np.pi/(nlons-1)
lats = (0.5*np.pi-delta*np.indices((nlats,nlons))[0,:,:])
lons = (delta*np.indices((nlats,nlons))[1,:,:])
wave = 0.75*(np.sin(2.*lats)**8*np.cos(4.*lons))
mean = 0.5*np.cos(2.*lats)*((np.sin(2.*lats))**2 + 2.)
# compute native map projection coordinates of lat/lon grid.
x, y = map(lons*180./np.pi, lats*180./np.pi)
# contour data over the map.
CS = map.contour(x,y,wave+mean,15,linewidths=1.5)
```

[](files/attachments/Matplotlib_Maps/basemap2b.png

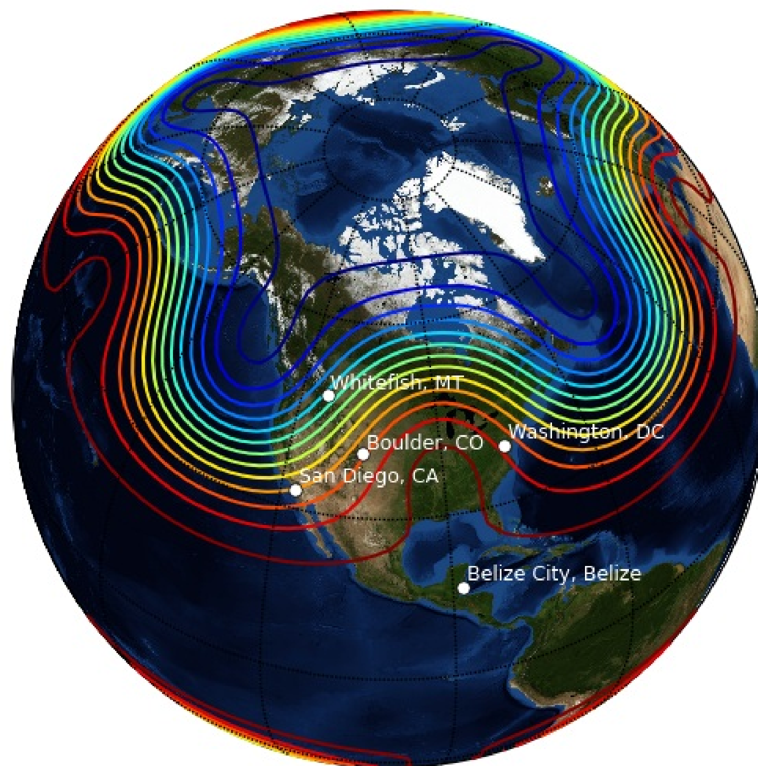
Instead of plotting the continents and coastlines, you can use an image as a map background using the method. The default background image is the NASA ‘blue marble’ image, which you can apply by using

```
map.bluemarble()
```

in place of

```
map.drawcoastlines()
map.drawcountries()
map.fillcontinents(color='coral')
```

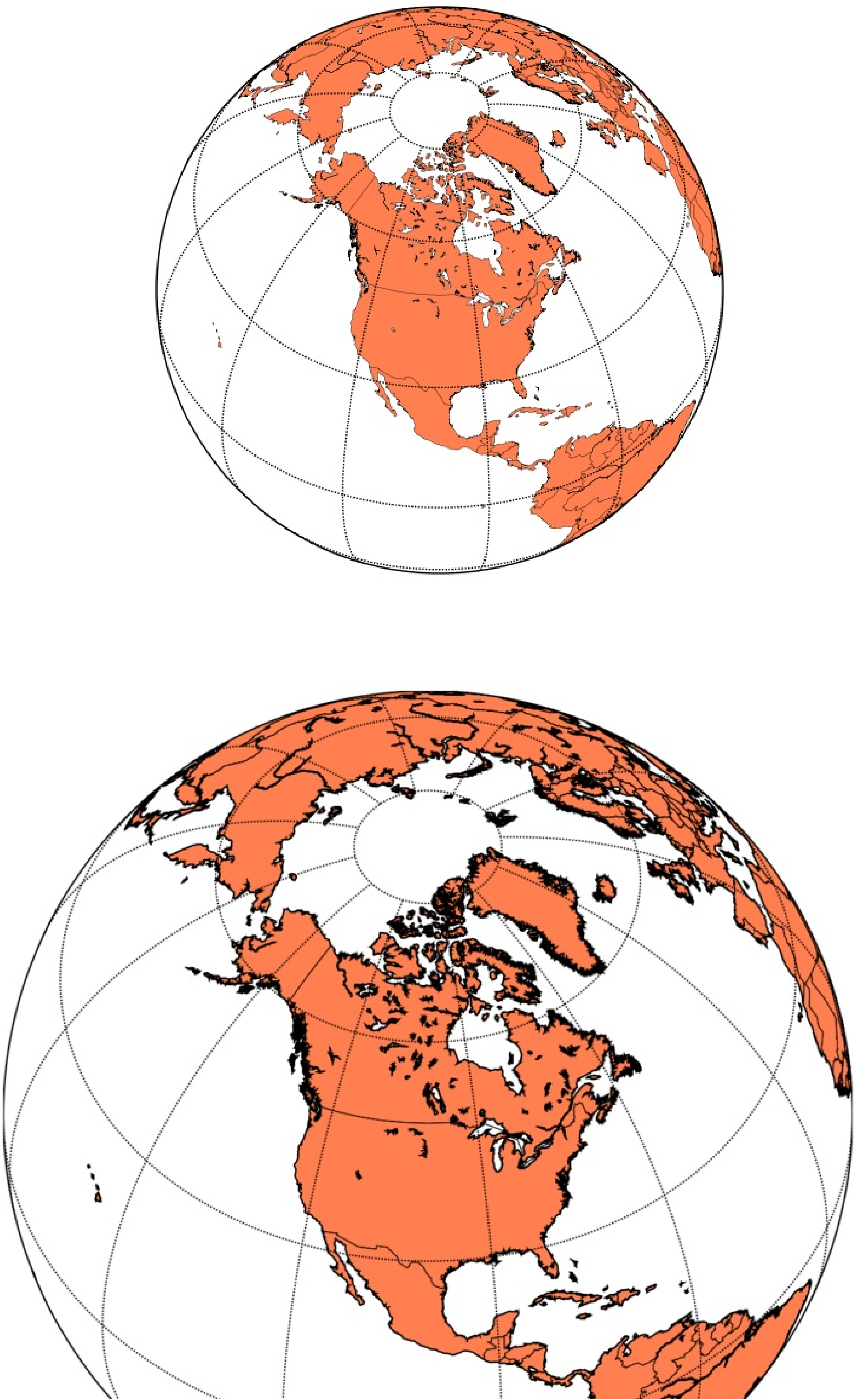
Here's what the resulting plot looks like (using white text instead of black, and white dots instead of blue)



You can also plot images, pcolor plots and vectors over map projections. Examples that illustrate this and more can be found in the examples directory of the basemap source distribution.

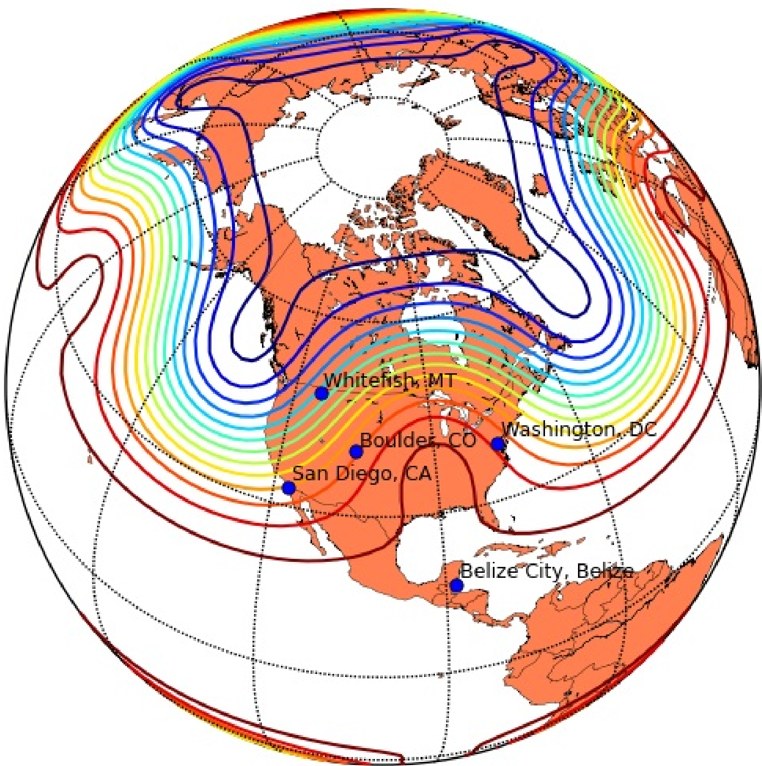
Attachments

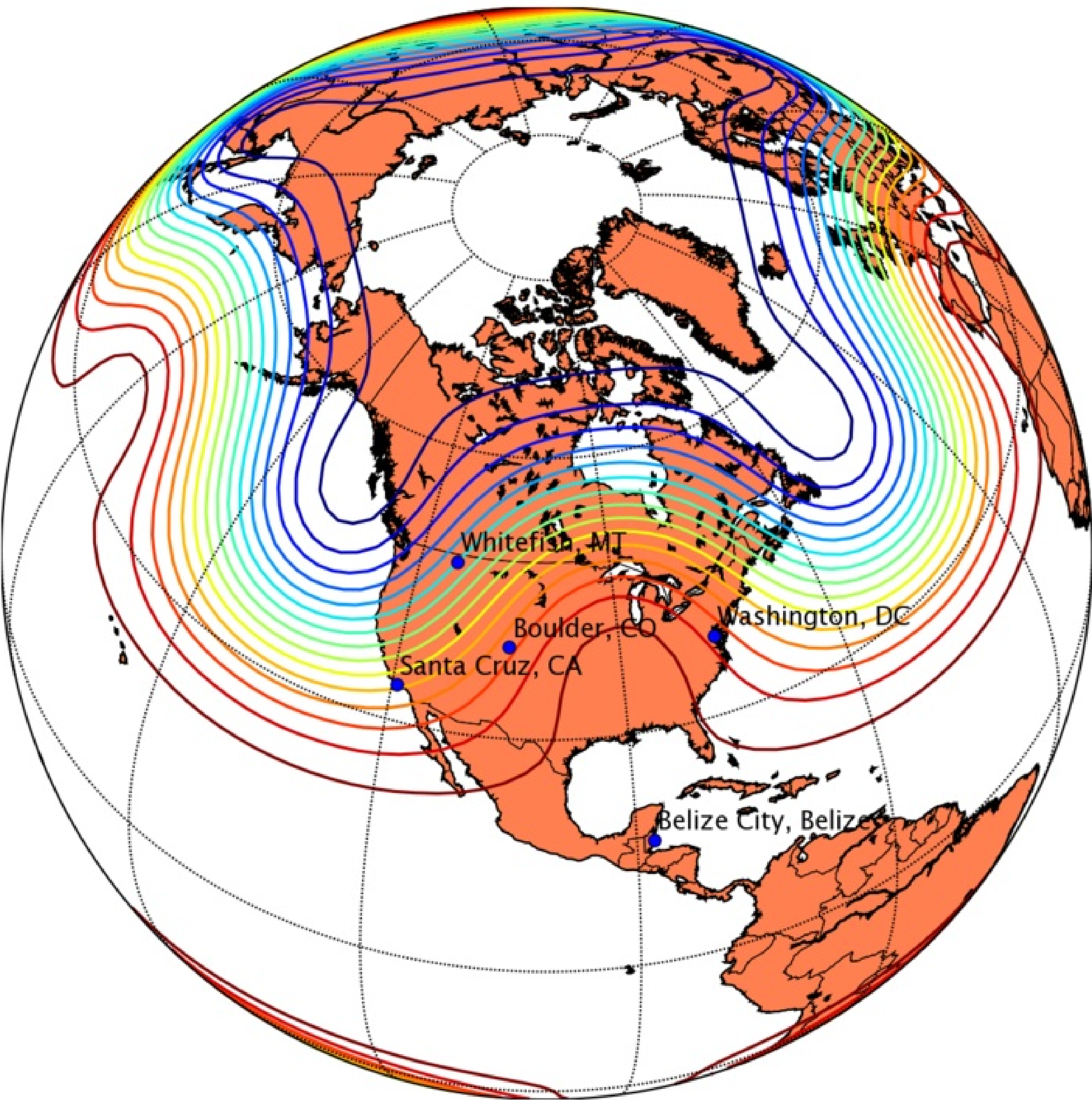
- `basemap0.png`
- `basemap1.png`
- `basemap1b.png`
- `basemap2.png`
- `basemap2b.png`
- `basemap3.png`
- `basemap3b.png`
- `basemap3c.png`

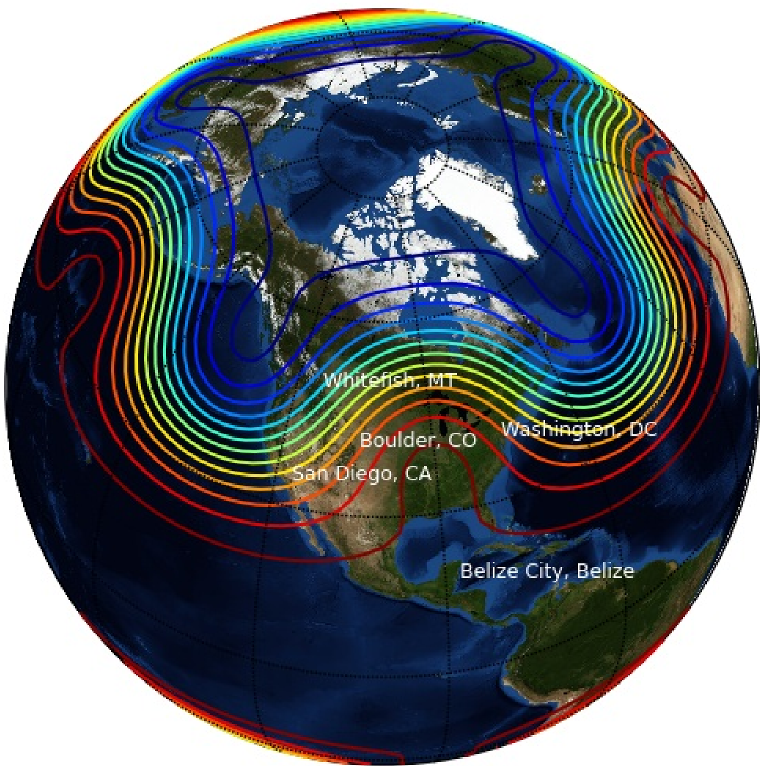


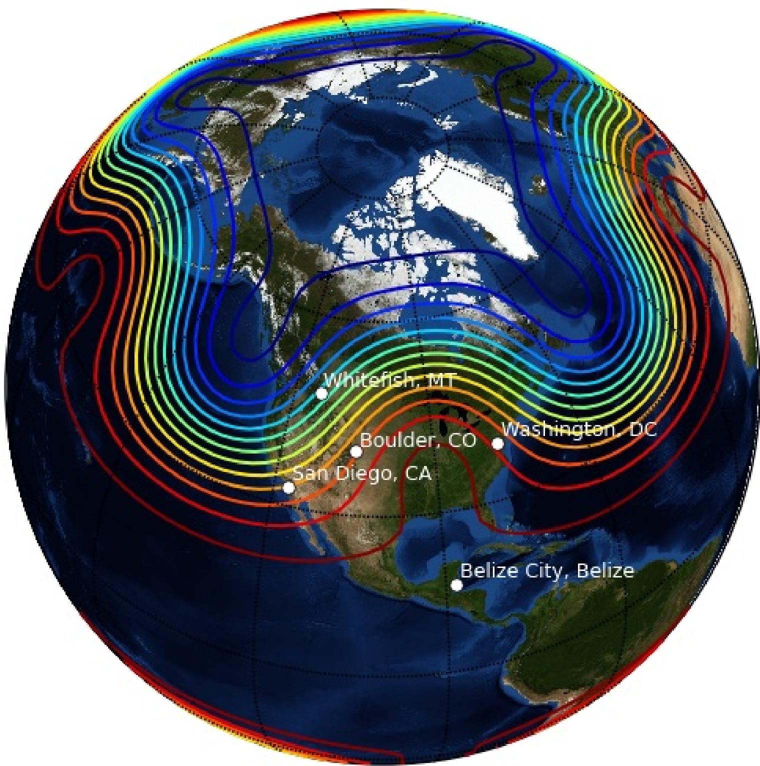

Whitefish, MT
Boulder, CO
Washington, DC
San Diego, CA
Belize City, Belize


Whitefish, MT
Boulder, CO
Washington, DC
Santa Cruz, CA
Belize City, Belize


Whitefish, MT
Boulder, CO
Washington, DC
San Diego, CA
Belize City, Belize


Whitefish, MT
Boulder, CO
Washington, DC
Santa Cruz, CA
Belize City, Belize


Whitefish, MT
Boulder, CO
Washington, DC
San Diego, CA
Belize City, Belize

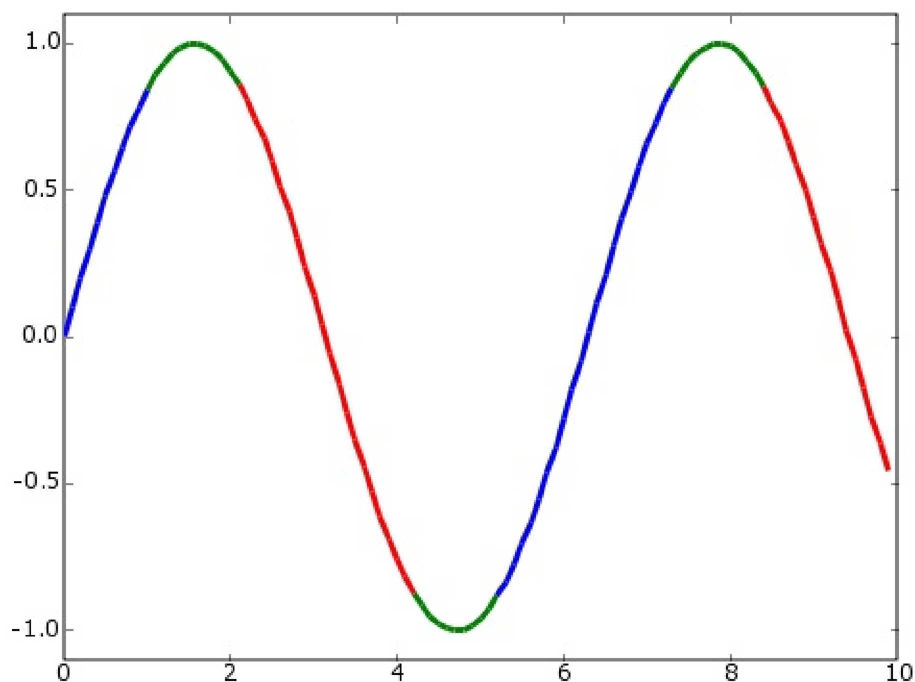

Whitefish, MT
Boulder, CO
Washington, DC
San Diego, CA
Belize City, Belize

Matplotlib: multicolored line

Defining colors manually

`colored_line.py` is a simple illustration of how to make the)# color of each segment of a line depend on some property of the data being plotted.

An up to date version of the script can be found here.



Here is the script:

```
#!/usr/bin/env python
'''
Color parts of a line based on its properties, e.g., slope.
'''
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from matplotlib.colors import ListedColormap, BoundaryNorm

x = np.linspace(0, 3 * np.pi, 500)
y = np.sin(x)
z = np.cos(0.5 * (x[:-1] + x[1:]))  # first derivative

# Create a colormap for red, green and blue and a norm to color
# f' < -0.5 red, f' > 0.5 blue, and the rest green
cmap = ListedColormap(['r', 'g', 'b'])
norm = BoundaryNorm([-1, -0.5, 0.5, 1], cmap.N)

# Create a set of line segments so that we can color them individua
# This creates the points as a N x 1 x 2 array so that we can stack
# together easily to get the segments. The segments array for line
# needs to be numlines x points per line x 2 (x and y)
points = np.array([x, y]).T.reshape(-1, 1, 2)
segments = np.concatenate([points[:-1], points[1:]], axis=1)

# Create the line collection object, setting the colormapping param
# Have to set the actual values used for colormapping separately.
lc = LineCollection(segments, cmap=cmap, norm=norm)
lc.set_array(z)
lc.set_linewidth(3)
plt.gca().add_collection(lc)

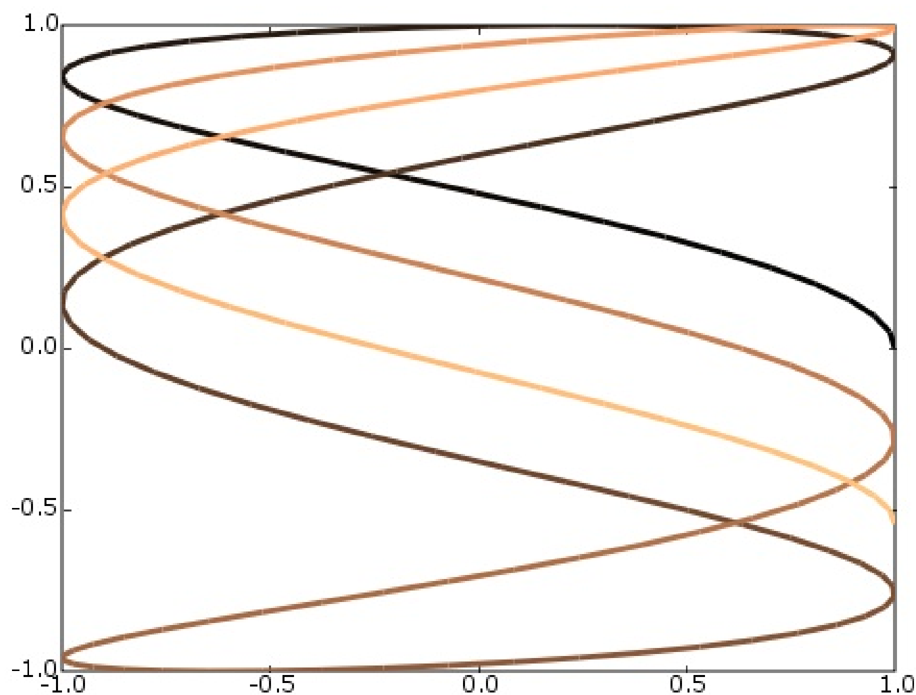
plt.xlim(x.min(), x.max())
plt.ylim(-1.1, 1.1)
plt.show()
```

Note that the number of segments is one less than the number of points.

An alternative strategy would be to generate one segment for each contiguous region of a given color.

Using a smooth, builtin colormap

If you have a parametric curve to display, and want to represent the parameter using color.




```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection

t = np.linspace(0, 10, 200)
x = np.cos(np.pi * t)
y = np.sin(t)

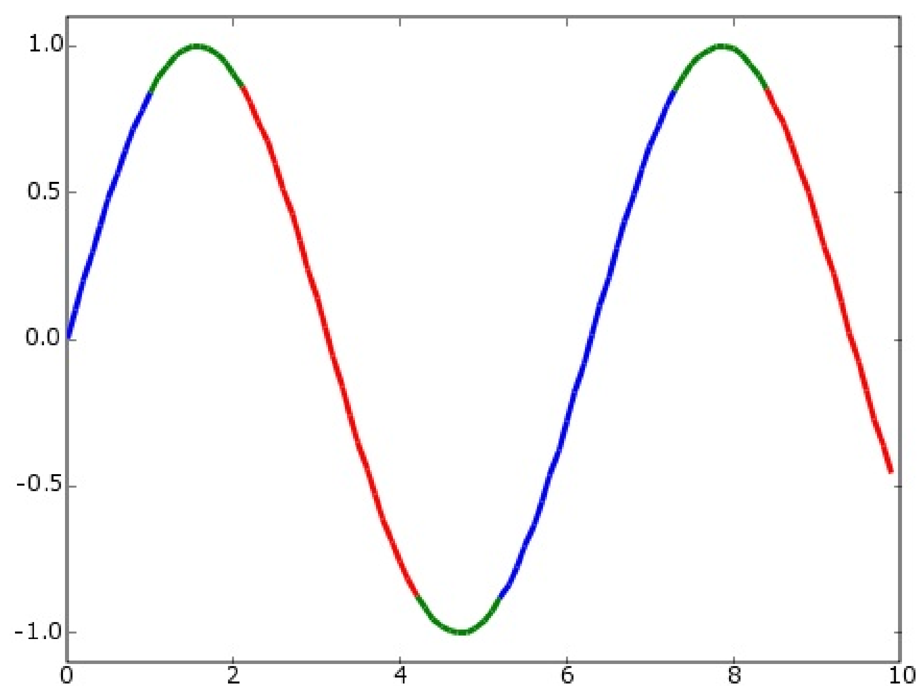
# Create a set of line segments so that we can color them individua
# This creates the points as a N x 1 x 2 array so that we can stack
# together easily to get the segments. The segments array for line
# needs to be numlines x points per line x 2 (x and y)
points = np.array([x, y]).T.reshape(-1, 1, 2)
segments = np.concatenate([points[:-1], points[1:]], axis=1)

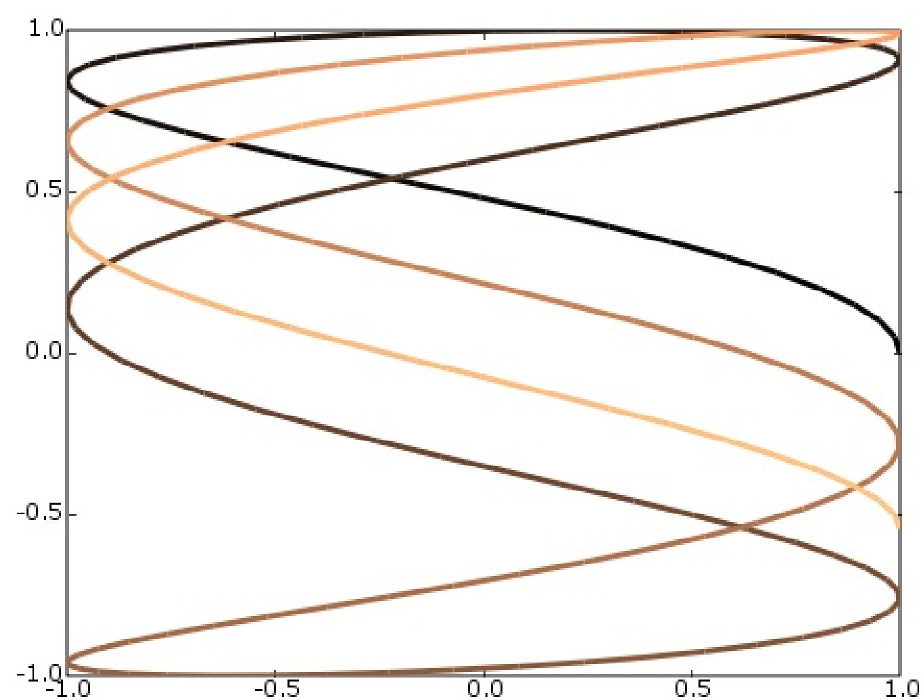
# Create the line collection object, setting the colormapping paran
# Have to set the actual values used for colormapping separately.
lc = LineCollection(segments, cmap=plt.get_cmap('copper'),
    norm=plt.Normalize(0, 10))
lc.set_array(t)
lc.set_linewidth(3)

plt.gca().add_collection(lc)
plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.show()
```

Attachments

- `colored_line.png`
- `colored_line.py`
- `colored_line2.png`





Matplotlib: multiline plots

Multiple line plots

Often one wants to plot many signals over one another. There are a few ways to do this. The naive implementation is just to add a constant offset to each signal:

```
from pylab import plot, show, ylim, yticks
from matplotlib.numerix import sin, cos, exp, pi, arange

t = arange(0.0, 2.0, 0.01)
s1 = sin(2*pi*t)
s2 = exp(-t)
s3 = sin(2*pi*t)*exp(-t)
s4 = sin(2*pi*t)*cos(4*pi*t)

t = arange(0.0, 2.0, 0.01)
plot(t, s1, t, s2+1, t, s3+2, t, s4+3, color='k')
ylim(-1,4)
yticks(arange(4), ['S1', 'S2', 'S3', 'S4'])

show()
```

but then it is difficult to do change the y scale in a reasonable way. For example when you zoom in on y, the signals on top and bottom will go off the screen. Often what one wants is for the y location of each signal to remain in place and the gain of the signal to be changed.

Using multiple axes

If you have just a few signals, you could make each signal a separate axes and make the y label horizontal. This works fine for a small number of signals (4-10 say) except the extra horizontal lines and ticks around the axes may be annoying. It’s on our list of things to change the way these axes lines are draw so that you can remove it, but it isn’t done yet:

```
from pylab import figure, show, setp
from matplotlib.numerix import sin, cos, exp, pi, arange

t = arange(0.0, 2.0, 0.01)
s1 = sin(2*pi*t)
s2 = exp(-t)
s3 = sin(2*pi*t)*exp(-t)
s4 = sin(2*pi*t)*cos(4*pi*t)

fig = figure()
t = arange(0.0, 2.0, 0.01)

yprops = dict(rotation=0,
              horizontalalignment='right',
              verticalalignment='center',
              x=-0.01)

axprops = dict(yticks=[])

ax1 =fig.add_axes([0.1, 0.7, 0.8, 0.2], **axprops)
ax1.plot(t, s1)
ax1.set_ylabel('S1', **yprops)

axprops['sharex'] = ax1
axprops['sharey'] = ax1
# force x axes to remain in register, even with toolbar navigation
ax2 = fig.add_axes([0.1, 0.5, 0.8, 0.2], **axprops)

ax2.plot(t, s2)
ax2.set_ylabel('S2', **yprops)

ax3 = fig.add_axes([0.1, 0.3, 0.8, 0.2], **axprops)
ax3.plot(t, s4)
ax3.set_ylabel('S3', **yprops)

ax4 = fig.add_axes([0.1, 0.1, 0.8, 0.2], **axprops)
ax4.plot(t, s4)
ax4.set_ylabel('S4', **yprops)

# turn off x ticklabels for all but the lower axes
for ax in ax1, ax2, ax3:
    setp(ax.get_xticklabels(), visible=False)

show()
```

[](files/attachments/Matplotlib_MultilinePlots/multipleaxes.png

Manipulating transforms

For large numbers of lines the approach above is inefficient because creating a separate axes for each line creates a lot of useless overhead. The application that gave birth to matplotlib is an EEG viewer which must efficiently handle hundreds of lines; this is is available as part of the pbrain package.

Here is an example of how that application does multiline plotting with “in place” gain changes. Note that this will break the y behavior of the toolbar because we have changed all the default transforms. In my application I have a custom toolbar to increase or decrease the y scale. In this example, I bind the plus/minus keys to a function which increases or decreases the y gain. Perhaps I will take this and wrap it up into a function called plot_signals or something like that because the code is a bit hairy since it makes heavy use of the somewhat arcane matplotlib transforms. I suggest reading up on the transforms module before trying to understand this example:

```
from pylab import figure, show, setp, connect, draw
from matplotlib.numerix import sin, cos, exp, pi, arange
from matplotlib.numerix.mlab import mean
from matplotlib.transforms import Bbox, Value, Point, \
     get_bbox_transform, unit_bbox
# load the data

t = arange(0.0, 2.0, 0.01)
s1 = sin(2*pi*t)
s2 = exp(-t)
s3 = sin(2*pi*t)*exp(-t)
s4 = sin(2*pi*t)*cos(4*pi*t)
s5 = s1*s2
s6 = s1-s4
s7 = s3*s4-s1

signals = s1, s2, s3, s4, s5, s6, s7
for sig in signals:
    sig = sig-mean(sig)

lineprops = dict(linewidth=1, color='black', linestyle='-')
fig = figure()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])

# The normal matplotlib transformation is the view lim bounding box
# (ax.viewLim) to the axes bounding box (ax.bbox).  Where are going
# define a new transform by defining a new input bounding box. See
# matplotlib.transforms module helkp for more information on
# transforms

# This bounding reuses the x data of the viewLim for the xscale -10
# 10 on the y scale.  -10 to 10 means that a signal with a min/max
# amplitude of 10 will span the entire vertical extent of the axes
scale = 10
boxin = Bbox(
    Point(ax.viewLim.ll().x(), Value(-scale)),
```

```
    Point(ax.viewLim.ur().x(), Value(scale)))

# height is a lazy value
height = ax.bbox.ur().y() - ax.bbox.ll().y()

boxout = Bbox(
    Point(ax.bbox.ll().x(), Value(-0.5) * height),
    Point(ax.bbox.ur().x(), Value( 0.5) * height))

# matplotlib transforms can accepts an offset, which is defined as
# point and another transform to map that point to display.  This
# transform maps x as identity and maps the 0-1 y interval to the
# vertical extent of the yaxis.  This will be used to offset the li
# and ticks vertically
transOffset = get_bbox_transform(
    unit_bbox(),
    Bbox( Point( Value(0), ax.bbox.ll().y()),
          Point( Value(1), ax.bbox.ur().y())
          ))

# now add the signals, set the transform, and set the offset of eac
# line
ticklocs = []
for i, s in enumerate(signals):
    trans = get_bbox_transform(boxin, boxout)
    offset = (i+1.)/(len(signals)+1.)
    trans.set_offset( (0, offset), transOffset)

    ax.plot(t, s, transform=trans, **lineprops)
    ticklocs.append(offset)

ax.set_yticks(ticklocs)
ax.set_yticklabels(['S%d'%(i+1) for i in range(len(signals))])

# place all the y tick attributes in axes coords
all = []
labels = []
ax.set_yticks(ticklocs)
for tick in ax.yaxis.get_major_ticks():
    all.extend(( tick.label1, tick.label2, tick.tick1line,
                 tick.tick2line, tick.gridline))
    labels.append(tick.label1)

setp(all, transform=ax.transAxes)
setp(labels, x=-0.01)

ax.set_xlabel('time (s)')

# Because we have hacked the transforms, you need a special method
# set the voltage gain; this is a naive implementation of how you
# might want to do this in real life (eg make the scale changes
# exponential rather than linear) but it gives you the idea
def set_ygain(direction):
```

```
    set_ygain.scale += direction
    if set_ygain.scale <=0:
        set_ygain.scale -= direction
        return

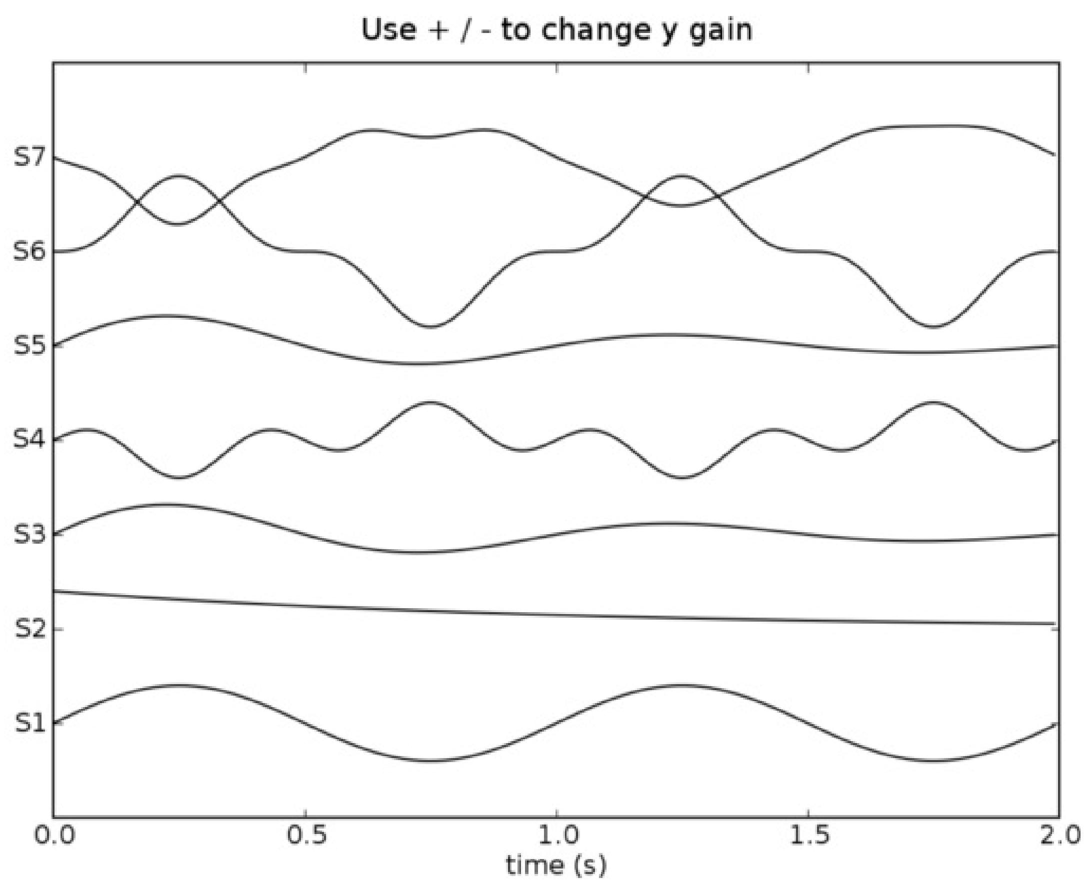
    for line in ax.lines:
        trans = line.get_transform()
        box1 =  trans.get_bbox1()
        box1.intervaly().set_bounds(-set_ygain.scale, set_ygain.sca
    draw()
set_ygain.scale = scale

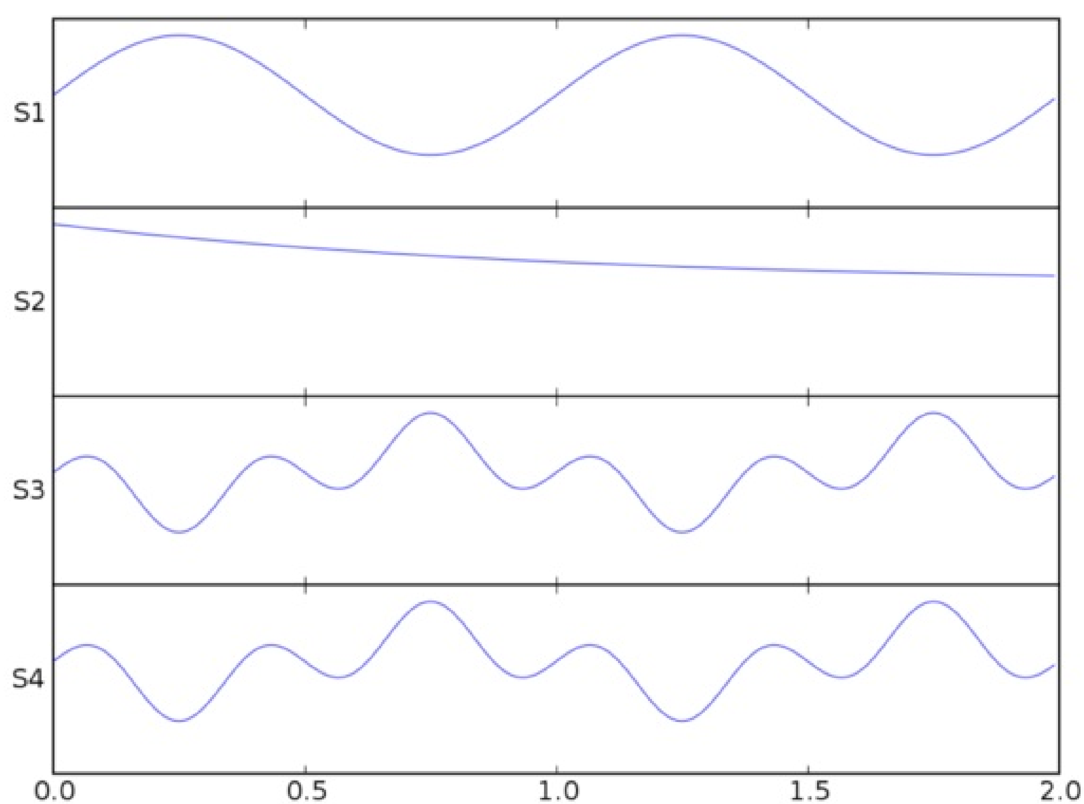
def keypress(event):
    if event.key in ('+', '='): set_ygain(-1)
    elif event.key in ('-', '_'): set_ygain(1)

connect('key_press_event', keypress)
ax.set_title('Use + / - to change y gain')
show()
```

Attachments

- `multiline.png`
- `multipleaxes.png`


Use + / - to change y gain
S7
S6
S5
S4
S3
S2
S1
0.0
0.5
1.0
1.5
2.0
time (s)


S1
S2
S3
S4
0.0
0.5
1.0
1.5
2.0

Matplotlib: plotting values with masked arrays

From time to time one might end up with “meaningless” data in an array. Be it because a detector didn’t work properly or for an other reason. Or one has to deal with data in completely different ranges. In both cases plotting all values will screw up the plot. This brief example script addresses this problem and show one possible solution using masked arrays. See ‘masked_demo.py’ in the matplotlib examples for a reference, too.

```
import numpy as np
import matplotlib.pyplot as plt

y_values = [0,0,100,97,98,0,99,101,0,102,99,105,101]
x_values = [0,1,2,3,4,5,6,7,8,9,10,11,12]

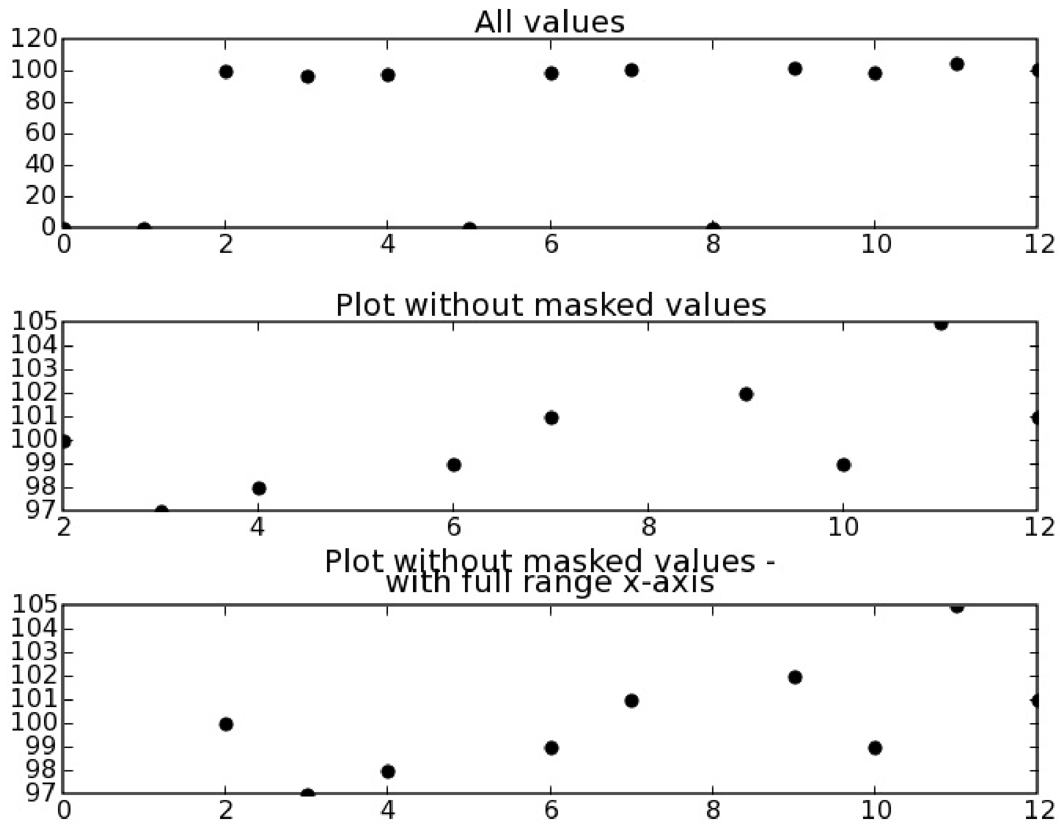
#give a threshold
threshold = 1

#prepare for masking arrays - 'conventional' arrays won't do it
y_values = np.ma.array(y_values)
#mask values below a certain threshold
y_values_masked = np.ma.masked_where(y_values < threshold , y_value

#plot all data
plt.subplots_adjust(hspace=0.5)
plt.subplot(311)
plt.plot(x_values, y_values,'ko')
plt.title('All values')
plt.subplot(312)
plt.plot(x_values, y_values_masked,'ko')
plt.title('Plot without masked values')
ax = plt.subplot(313)
ax.plot(x_values, y_values_masked,'ko')
#for otherwise the range of x_values gets truncated:
ax.set_xlim(x_values[0], x_values[-1])
plt.title('Plot without masked values -\nwith full range x-axis')

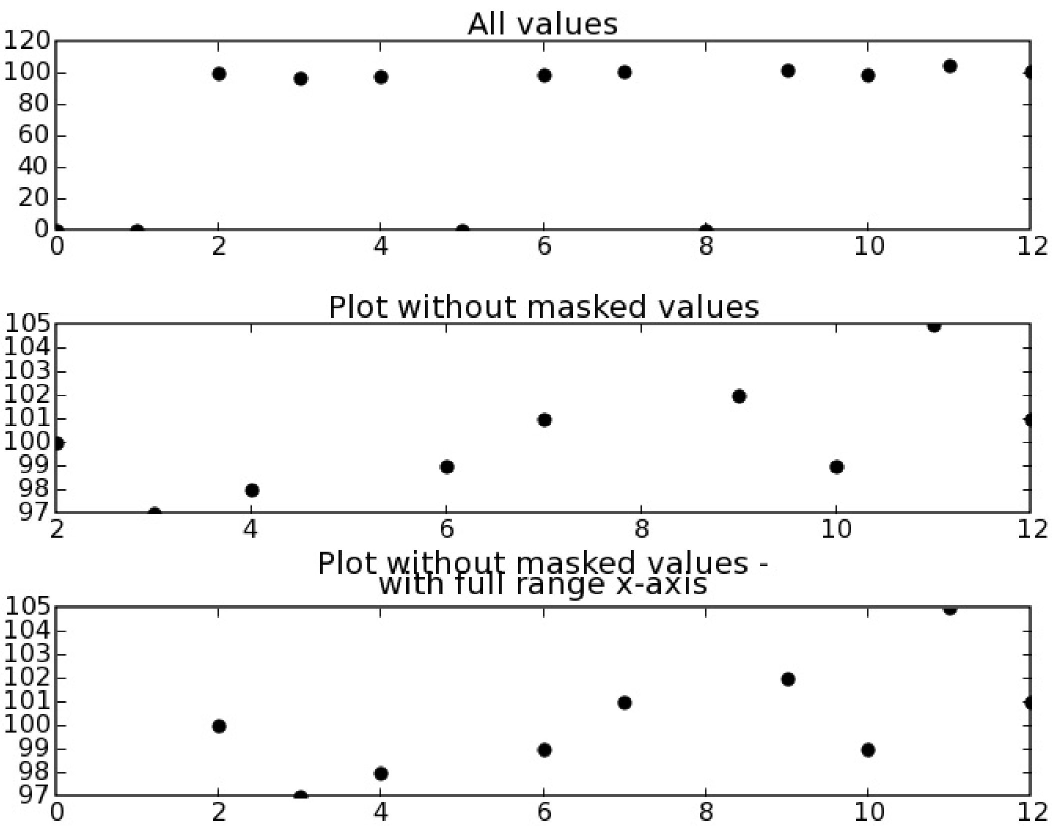
savefig('masked_test.png')
```

The resulting figure might illustrate the problem - note the different scales in all three subplots:



Attachments

- `masked_test.png`


All values
120
100
80
60
40
20
0
0
2
4
6
8
10
12
Plot without masked values
105
104
103
102
101
100
99
98
97
2
4
6
8
10
12
Plot without masked values -
with full range x-axis
105
104
103
102
101
100
99
98
97
0
2
4
6
8
10
12

Matplotlib: shaded regions

Use the fill function to make shaded regions of any color tint. Here is an example.

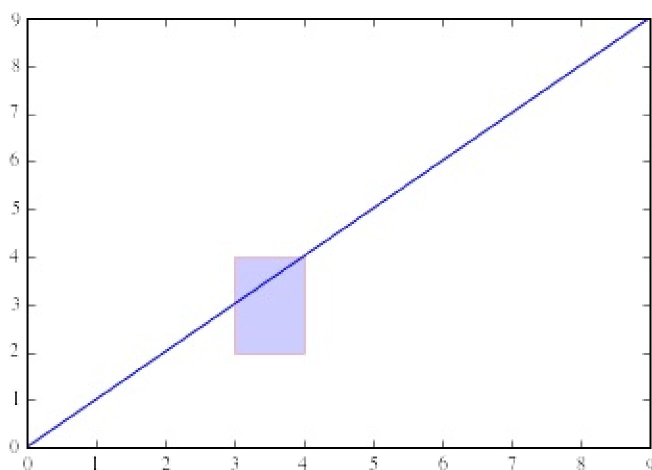
```
from pylab import *

x = arange(10)
y = x

# Plot junk and then a filled region
plot(x, y)

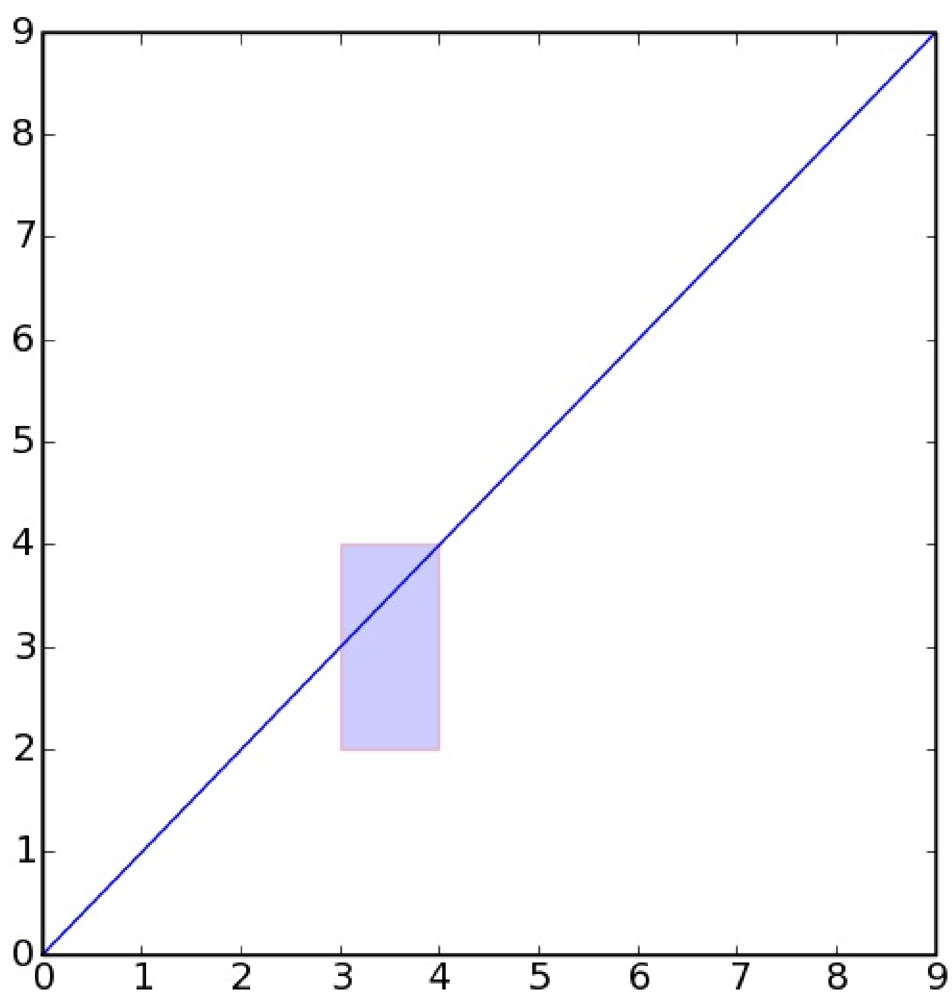
# Make a blue box that is somewhat see-through
# and has a red border.
# WARNING: alpha doesn't work in postscript output....
fill([3,4,4,3], [2,2,4,4], 'b', alpha=0.2, edgecolor='r')
```

```
[<matplotlib.patches.Polygon at 0x7f7a19aac890>]
```



Attachments

- `shaded.png`


9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9

Matplotlib: sigmoidal functions

matplotlib's approach to plotting functions requires you to compute the x and y vertices of the curves you want to plot and then pass it off to plot. Eg for a normal pdf, matplotlib.mlab provides such a function:

```
from matplotlib.mlab import normpdf
import matplotlib.numerix as nx
import pylab as p

x = nx.arange(-4, 4, 0.01)
y = normpdf(x, 0, 1) # unit normal
p.plot(x,y, color='red', lw=2)
p.show()
```

Of course, some curves do not have closed form expressions and are not amenable for such treatment. Some of the matplotlib backends have the capability to draw arbitrary paths with splines (cubic and quartic) but this functionality hasn't been exposed to the user yet (as of 0.83). If you need this, please post to the mailing list or submit a sourceforge support request.

Rich Shepard was interested in plotting "S curves" and "Z curves", and a little bit of googling suggests that the S curve is a sigmoid and the Z curve is simply 1.0-sigmoid. There are many simple forms for sigmoids: eg, the hill, boltzman, and arc tangent functions. Here is an example of the boltzman function:

```
import matplotlib.numerix as nx
import pylab as p

def boltzman(x, xmid, tau):
    """
 evaluate the boltzman function with midpoint xmid and time constan
 over x
 """
    return 1. / (1. + nx.exp(-(x-xmid)/tau))

x = nx.arange(-6, 6, .01)
S = boltzman(x, 0, 1)
Z = 1-boltzman(x, 0.5, 1)
p.plot(x, S, x, Z, color='red', lw=2)
p.show()
```

See also sigmoids at mathworld.

People often want to shade an area under these curves, eg under their intersection, which you can do with the magic of numerix and the matplotlib [http://matplotlib.sourceforge.net/matplotlib.pylab.html#-fill fill] function:

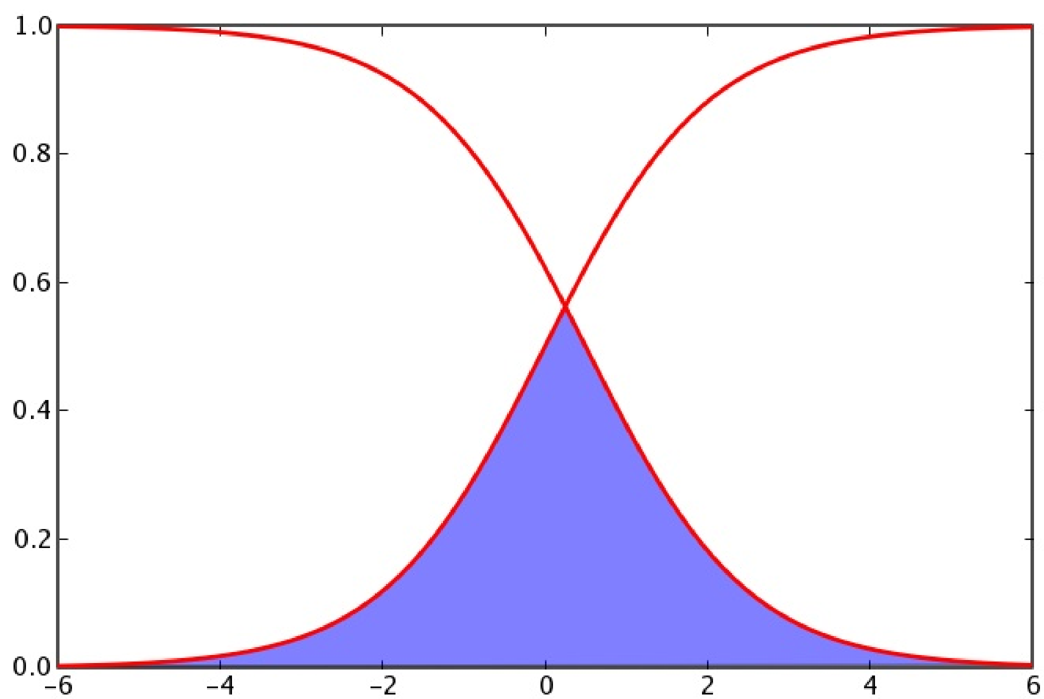
```
import matplotlib.numerix as nx
import pylab as p

def boltzman(x, xmid, tau):
    """
 evaluate the boltzman function with midpoint xmid and time constar
 over x
 """
    return 1. / (1. + nx.exp(-(x-xmid)/tau))

def fill_below_intersection(x, S, Z):
    """
 fill the region below the intersection of S and Z
 """
    #find the intersection point
    ind = nx.nonzero( nx.absolute(S-Z)==min(nx.absolute(S-Z)))[0]
    # compute a new curve which we will fill below
    Y = nx.zeros(S.shape, typecode=nx.Float)
    Y[:ind] = S[:ind]  # Y is S up to the intersection
    Y[ind:] = Z[ind:]  # and Z beyond it
    p.fill(x, Y, facecolor='blue', alpha=0.5)

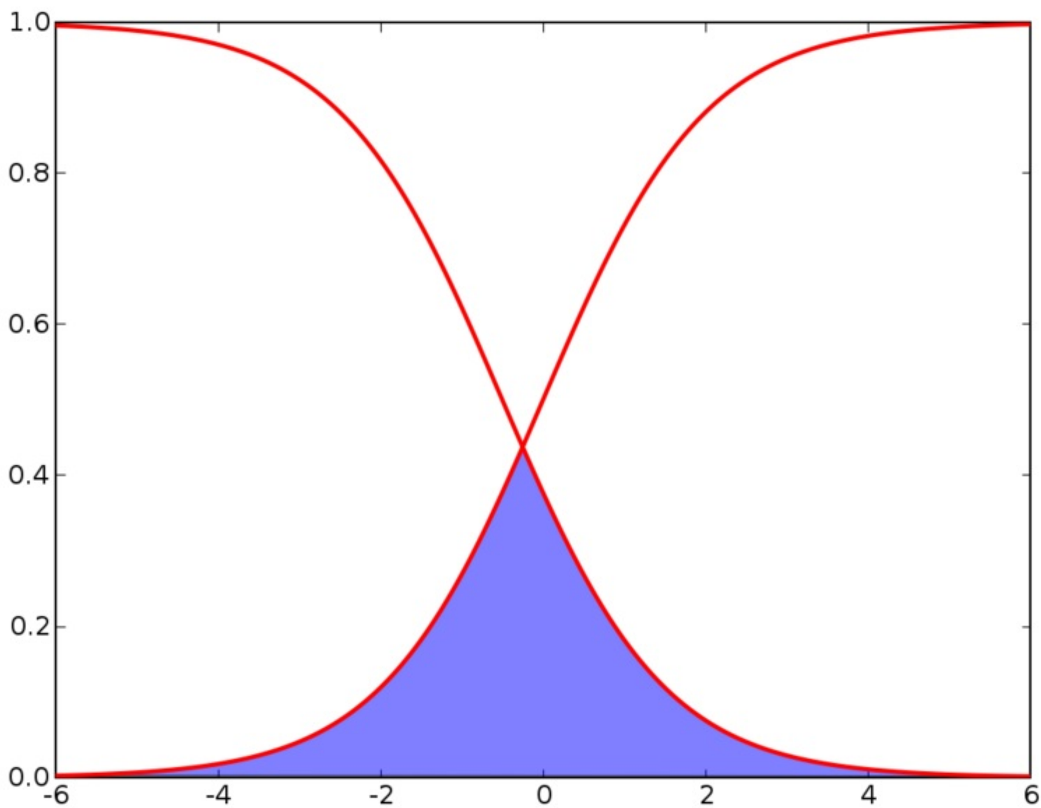
x = nx.arange(-6, 6, .01)
S = boltzman(x, 0, 1)
Z = 1-boltzman(x, 0.5, 1)
p.plot(x, S, x, Z, color='red', lw=2)
fill_below_intersection(x, S, Z)
p.show()
```

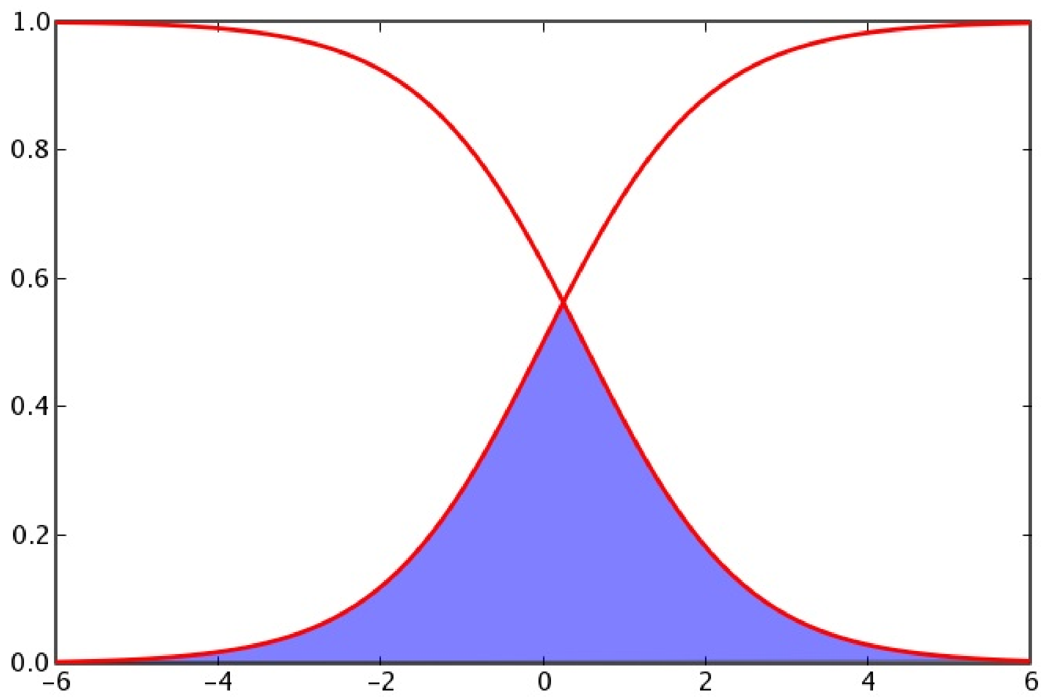
As these examples illustrate, matplotlib doesn’t come with helper functions for all the kinds of curves people want to plot, but along with numerix and python, provides the basic tools to enable you to build them yourself.



Attachments

- `sigmoids.png`
- `sigmoids2.png`


1.0
0.8
0.6
0.4
0.2
0.0
-6
-4
-2
0
2
4
6


1.0
0.8
0.6
0.4
0.2
0.0
−6
−4
−2
0
2
4
6

Matplotlib: thick axes

Example of how to thicken the lines around your plot (axes lines) and to get big bold fonts on the tick and axis labels.

```
from pylab import *

# Thicken the axes lines and labels
#
#   Comment by J. R. Lu:
#       I couldn't figure out a way to do this on the
#       individual plot and have it work with all backends
#       and in interactive mode. So, used rc instead.
#
rc('axes', linewidth=2)

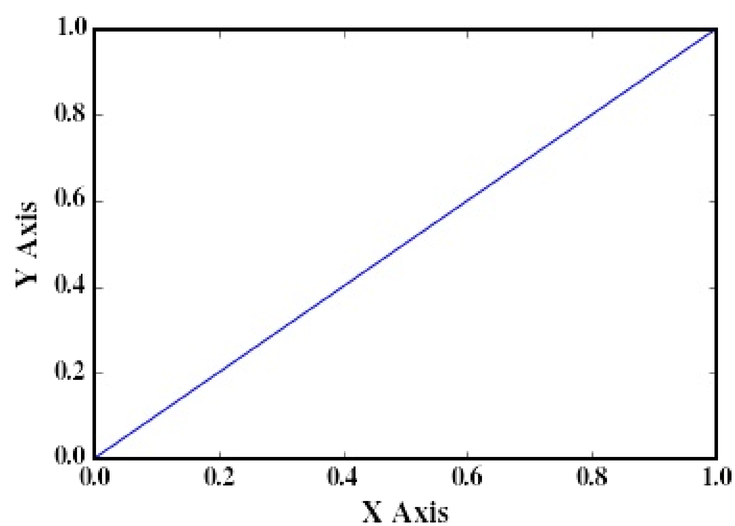
# Make a dummy plot
plot([0, 1], [0, 1])

# Change size and font of tick labels
# Again, this doesn't work in interactive mode.
fontsize = 14
ax = gca()

for tick in ax.xaxis.get_major_ticks():
    tick.label1.set_fontsize(fontsize)
    tick.label1.set_fontweight('bold')
for tick in ax.yaxis.get_major_ticks():
    tick.label1.set_fontsize(fontsize)
    tick.label1.set_fontweight('bold')

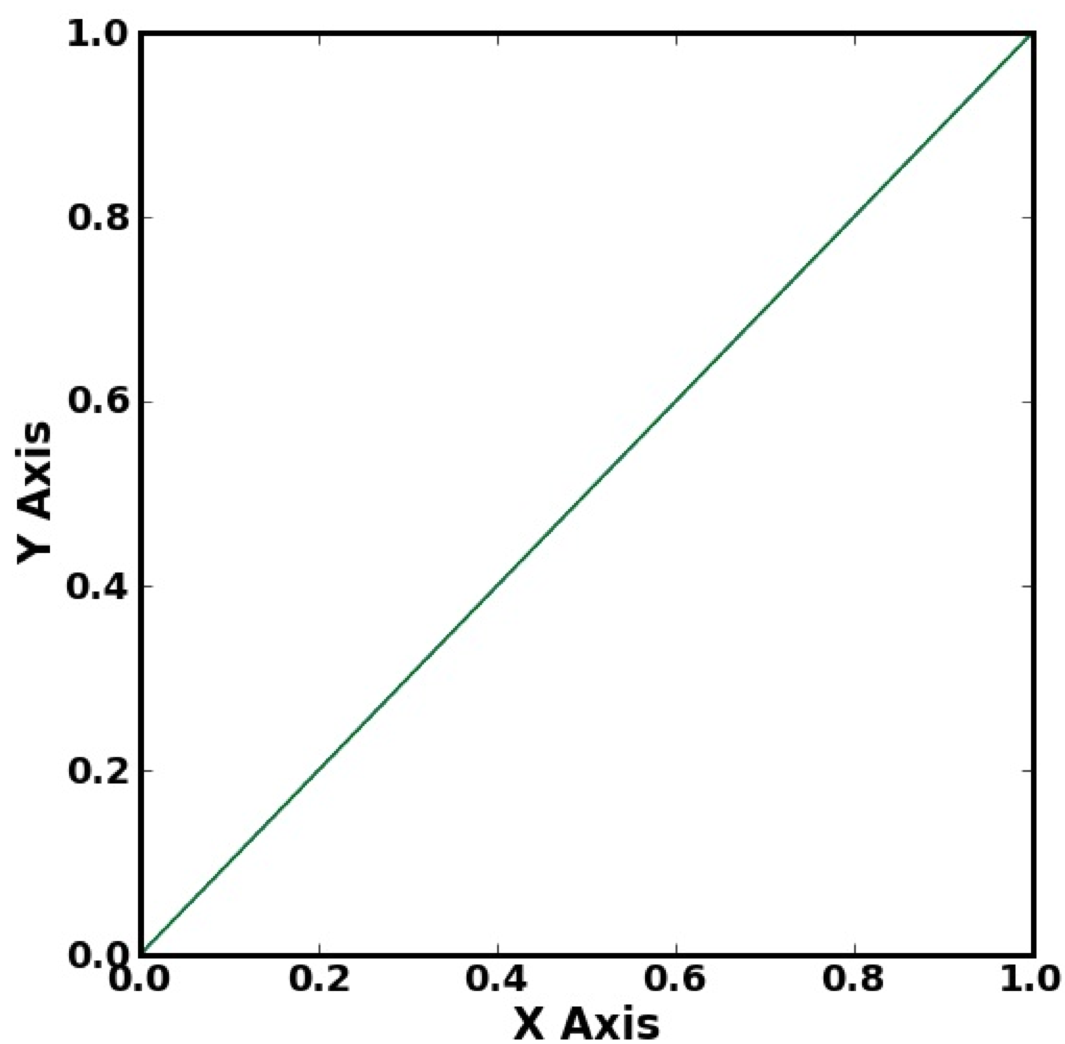
xlabel('X Axis', fontsize=16, fontweight='bold')
ylabel('Y Axis', fontsize=16, fontweight='bold')

# Save figure
savefig('thick_axes.png')
```



Attachments

- `thick_axes.png`



Matplotlib: transformations

Whenever you pass coordinates to matplotlib, the question arises, what kind of coordinates you mean. Consider the following example

```
axes.text(x,y, "my label")
```

A label 'my label' is added to the axes at the coordinates x,y, or stated more clearly: The text is placed at the theoretical position of a data point (x,y). Thus we would speak of "data coords". There are however other coordinates one can think of. You might e.g. want to put a label in the exact middle of your graph. If you specified this by the method above, then you would need to determine the minimum and maximum values of x and y to determine the middle. However, using transforms, you can simply use

```
axes.text(0.5, 0.5, "middle of graph", transform=axes.transAxes)
```

There are four built-in transforms that you should be aware of (let ax be an Axes instance and fig a Figure instance):

```
matplotlib.transforms.identity_transform()  # display coords
ax.transData     # data coords
ax.transAxes     # 0,0 is bottom,left of axes and 1,1 is top,right
fig.transFigure  # 0,0 is bottom,left of figure and 1,1 is top,righ
```

These transformations can be used for any kind of Artist, not just for text objects.

The default transformation for ax.text is ax.transData and the default transformation for fig.text is fig.transFigure.

Of course, you can define more general transformations, e.g. matplotlib.transforms.Affine, but the four listed above arise in a lot of applications.

xy_tup() is no more. Please see the official Matplotlib documentation at http://matplotlib.sourceforge.net/users/transforms_tutorial.html for further reference.

Example: tick label like annotations

If you find that the built-in tick labels of Matplotlib are not enough for you, you can use transformations to implement something similar. Here is an example that draws annotations below the tick labels, and uses a transformation to guarantee

that the x coordinates of the annotation correspond to the x coordinates of the plot, but the y coordinates are at a fixed position, independent of the scale of the plot:

```
import matplotlib as M
import Numeric as N
import pylab as P
blend = M.transforms.blend_xy_sep_transform

def doplot(fig, subplot, function):
    ax = fig.add_subplot(subplot)
    x = N.arange(0, 2*N.pi, 0.05)
    ax.plot(x, function(x))

    trans = blend(ax.transData, ax.transAxes)

    for x,text in [(0.0, '|'), (N.pi/2, r'$\rm{zero\ to\ }\pi$'),
                   (N.pi, '|'), (N.pi*1.5, r'$\pi\rm{\ to\ }2\pi$')
                   (2*N.pi, '|')]:
        ax.text(x, -0.1, text, transform=trans,
                horizontalalignment='center')

fig = P.figure()
doplot(fig, 121, N.sin)
doplot(fig, 122, lambda x: 10*N.sin(x))
P.show()
```

Example: adding a pixel offset to data coords

Sometimes you want to specify that a label is shown a fixed *pixel* offset from the corresponding data point, regardless of zooming. Here is one way to do it; try running this in an interactive backend, and zooming and panning the figure.

The way this works is by first taking a shallow copy of `transData` and then adding an offset to it. All transformations can have an offset which can be modified with `set_offset`, and the copying is necessary to avoid modifying the transform of the data itself. New enough versions of matplotlib (currently only the svn version) have an `offset_copy` function which does this automatically.

```
import matplotlib
import matplotlib.transforms
from pylab import figure, show

# New enough versions have offset_copy by Eric Firing:
if 'offset_copy' in dir(matplotlib.transforms):
    from matplotlib.transforms import offset_copy
    def offset(ax, x, y):
        return offset_copy(ax.transData, x=x, y=y, units='dots')
else: # Without offset_copy we have to do some black transform magi
    from matplotlib.transforms import blend_xy_sep_transform, ident
    def offset(ax, x, y):
        # This trick makes a shallow copy of ax.transData (but fail
        trans = blend_xy_sep_transform(ax.transData, ax.transData)
        # Now we set the offset in pixels
        trans.set_offset((x,y), identity_transform())
        return trans

fig=figure()
ax=fig.add_subplot(111)

# plot some data
x = (3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3)
y = (2,7,1,8,2,8,1,8,2,8,4,5,9,0,4,5)
ax.plot(x,y,'.')

# add labels
trans=offset(ax, 10, 5)
for a,b in zip(x,y):
    ax.text(a, b, '(%d,%d)'%(a,b), transform=trans)

show()
```

Matplotlib: unfilled histograms

Here’s some template code for plotting histograms that don’t look like bar charts, but instead have only outlines (like IDL creates).

First define a function that does the bulk of the heavy lifting.

```
import numpy as np

def histOutline(dataIn, *args, **kwargs):
    (histIn, binsIn) = np.histogram(dataIn, *args, **kwargs)

    stepSize = binsIn[1] - binsIn[0]

    bins = np.zeros(len(binsIn)*2 + 2, dtype=np.float)
    data = np.zeros(len(binsIn)*2 + 2, dtype=np.float)
    for bb in range(len(binsIn)):
        bins[2*bb + 1] = binsIn[bb]
        bins[2*bb + 2] = binsIn[bb] + stepSize
        if bb < len(histIn):
            data[2*bb + 1] = histIn[bb]
            data[2*bb + 2] = histIn[bb]

    bins[0] = bins[1]
    bins[-1] = bins[-2]
    data[0] = 0
    data[-1] = 0

    return (bins, data)
```

Now we can make plots:

```
# Make some data to plot
data = randn(500)

figure(2, figsize=(10, 5))
clf()

##########
#
# First make a normal histogram
#
##########
subplot(1, 2, 1)
(n, bins, patches) = hist(data)

# Boundaries
xlo = -max(abs(bins))
xhi = max(abs(bins))
ylo = 0
yhi = max(n) * 1.1

axis([xlo, xhi, ylo, yhi])

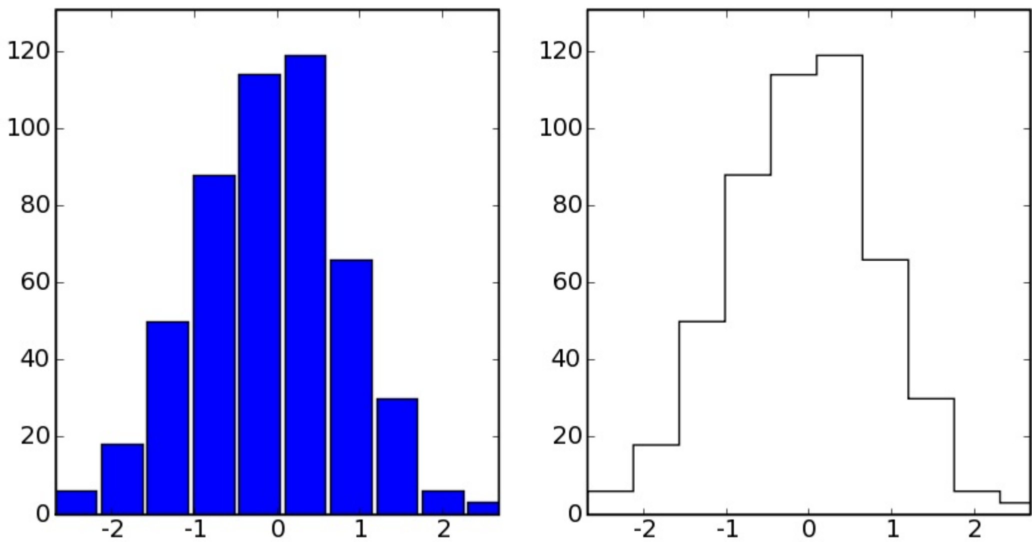
##########
#
# Now make a histogram in outline format
#
##########
(bins, n) = histOutline(data)

subplot(1, 2, 2)
plot(bins, n, 'k-')
axis([xlo, xhi, ylo, yhi])
```

Below you can find this functionality packaged up into histOutline.py

Attachments

- `histNofill.py`
- `histOutline.py`
- `hist_outline.png`


120
100
80
60
40
20
0
-2
-1
0
1
2
120
100
80
60
40
20
0
-2
-1
0
1
2

Matplotlib / Typesetting

Matplotlib: latex examples

Producing Graphs for Publication using LaTeX

This page describes several ways to produce publication quality graphics with LaTeX.

LaTeX UsingTex

This section describes a technique following the [“Cookbook/Matplotlib/UsingTex”] guidelines.

Here is the outline of the LaTeX file used to include the figure (example for `REVTeX4` for publication is APS physics journals with a two column format.)

```
\documentclass[prl,10pt,twocolumn]{revtex4}
\usepackage{graphicx}    % Used to import the graphics
\begin{document}
%...
\begin{figure}[t]
  \begin{center}
    \showthe\columnwidth % Use this to determine the width of the f
    \includegraphics[width=\columnwidth]{fig1.eps}
    \caption{\label{fig:sin_cos} Plot of the sine and cosine functi
  \end{center}
\end{figure}
%...
\end{document}
```

Determining the Figure Size

The first step is to determine the size of the figure: this way, when the graphic is included, it will not be resized, and the fonts etc. will be exactly as you set them rather than scaled (and possibly distored). This can be done in LaTeX by explicitly setting the width of the figure and using the `\showthe` command to print this width. (In the example above, the figure width is set to the `\columnwidth` .)

When the file is processed by LaTeX, look at the output. The example above produces the following output (Note: LaTeX will pause after the `\showthe` command, press enter to continue):

```
This is TeX, Version 3.14159 (Web2C 7.4.5)
LaTeX2e <2001/06/01>
...
> 246.0pt.
l.8     \showthe\columnwidth
                             % Use this to determine the width of t
?
<fig1.eps> [1] (./tst.aux) )
...
```

Thus, the figure will be 246.0pt wide. There are 1 inch = 72.27pt (in [La]TeX), so this means that the figure width should be 3.40390 inches. The heigth depends on the content of the figure, but the golden mean may be used to make a pleasing figure. Once this is determined, the `figure.figsize` property can be used to set the default figure size.

```
#!python numbers=disable
fig_width_pt = 246.0  # Get this from LaTeX using \showthe\columnwi
inches_per_pt = 1.0/72.27               # Convert pt to inches
golden_mean = (sqrt(5)-1.0)/2.0         # Aesthetic ratio
fig_width = fig_width_pt*inches_per_pt  # width in inches
fig_height =fig_width*golden_mean       # height in inches
fig_size = [fig_width,fig_height]
```

Setting Font Sizes

Since the figure will not be scaled down, we may explicitly set the font sizes.

```
#!python numbers=disable
          'font.size' : 10,
          'axes.labelsize' : 10,
          'font.size' : 10,
          'text.fontsize' : 10,
          'legend.fontsize': 10,
          'xtick.labelsize' : 8,
          'ytick.labelsize' : 8,
```

Fine Tuning

With these smaller plot sizes, the default margins are not enough to display the axis labels, so we need to specify large margins. We do this with an explicit call to the `axes()` function. In this example, we only have one axis. The typeset LaTeX document will have whitespace on either side of the figure, so we do not need to

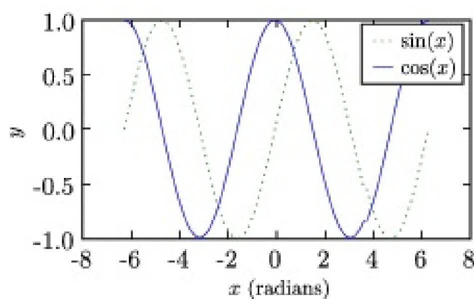
include this in the figure. Thus, we keep just a bit of whitespace at the top and to the right so that the labels do not extend beyond the bounding box, and add more space to the bottom for the x label:

```
#!python numbers=disable
pylab.axes([0.125,0.2,0.95-0.125,0.95-0.2])
```

Putting it all Together

Here is the python file that generates the plots.

```
#!python
import pylab
from pylab import arange,pi,sin,cos,sqrt
fig_width_pt = 246.0  # Get this from LaTeX using \showthe\columnwi
inches_per_pt = 1.0/72.27               # Convert pt to inch
golden_mean = (sqrt(5)-1.0)/2.0         # Aesthetic ratio
fig_width = fig_width_pt*inches_per_pt  # width in inches
fig_height = fig_width*golden_mean      # height in inches
fig_size =  [fig_width,fig_height]
params = {'backend': 'ps',
          'axes.labelsize': 10,
          'text.fontsize': 10,
          'legend.fontsize': 10,
          'xtick.labelsize': 8,
          'ytick.labelsize': 8,
          'text.usetex': True,
          'figure.figsize': fig_size}
pylab.rcParams.update(params)
# Generate data
x = pylab.arange(-2*pi,2*pi,0.01)
y1 = sin(x)
y2 = cos(x)
# Plot data
pylab.figure(1)
pylab.clf()
pylab.axes([0.125,0.2,0.95-0.125,0.95-0.2])
pylab.plot(x,y1,'g:',label='$\sin(x)$')
pylab.plot(x,y2,'-b',label='$\cos(x)$')
pylab.xlabel('$x$ (radians)')
pylab.ylabel('$y$')
pylab.legend()
pylab.savefig('fig1.eps')
```



producing gray scale dashed plots

```
An obvious solution is to greyscale convert your figure, but for rea
```

LaTeX using psfrag

Note: This section is obsolete. Recent versions matplotlib break the psfrag functionality (see for example this discussion. That being said, one can use the usetex feature to render the LaTeX text directly with very good results (if you are careful about choosing fonts). I will try to discuss this here further in the near future. – MichaelMcNeilForbes

To ensure that your graphics use exactly the same fonts as your document, you can have LaTeX generate and substitute the text for your graph using the psfrag package. This is a good option if you have problems with the `text.usetex` method (for example, if the appropriate fonts cannot be found.)

To do this, simply use plain text for the labels and then replace them using the psfrag package. Here are the modified files to make use of this method:

```
\documentclass[prl,10pt,twocolumn]{revtex4}
\usepackage{graphicx}    % Used to import the graphics
\usepackage{psfrag}
\begin{document}
%...
\begin{figure}[t]
  \begin{center}
    \psfrag{sin(x)}{$\sin(x)$}
    \psfrag{cos(x)}{$\cos(x)$}
    \psfrag{x (radians)}{$x$ (radians)}
    \psfrag{y}{$y$}
    {\footnotesize                  % Replace tick-lables with smal
      \psfrag{1.0}{1.0}
      \psfrag{0.5}{0.5}
      \psfrag{0.0}{0.0}
      \psfrag{-0.5}{-0.5}
      \psfrag{-1.0}{-1.0}
      \psfrag{-8}{-8}
      \psfrag{-6}{-6}
      \psfrag{-4}{-4}
      \psfrag{-2}{-2}
      \psfrag{0}{0}
      \psfrag{2}{2}
      \psfrag{4}{4}
      \psfrag{6}{6}
      \psfrag{8}{8}
      \showthe\columnwidth % Use this to determine the width of the
      \includegraphics[width=\columnwidth]{fig2.eps}
    } % Note that the psfrag commands only work in the top-most env
    \caption{\label{fig:sin_cos} Plot of the sine and cosine functi
  \end{center}
\end{figure}
%...
\end{document}
```

```
#!python
import pylab
from pylab import arange,pi,sin,cos,sqrt
fig_width_pt = 246.0  # Get this from LaTeX using \showthe\columnwi
inches_per_pt = 1.0/72.27               # Convert pt to inch
golden_mean = (sqrt(5)-1.0)/2.0         # Aesthetic ratio
fig_width = fig_width_pt*inches_per_pt  # width in inches
fig_height = fig_width*golden_mean      # height in inches
fig_size =  [fig_width,fig_height]
params = {'backend': 'ps',
          'axes.labelsize': 10,
          'text.fontsize': 10,
          'legend.fontsize': 10,
          'xtick.labelsize': 8,
          'ytick.labelsize': 8,
          'text.usetex': False,
          'figure.figsize': fig_size}
pylab.rcParams.update(params)
# Generate data
x = pylab.arange(-2*pi,2*pi,0.01)
y1 = sin(x)
y2 = cos(x)
# Plot data
# Plot data
pylab.figure(1)
pylab.clf()
pylab.axes([0.125,0.2,0.95-0.125,0.95-0.2])
pylab.plot(x,y1,'g:',label='sin(x)')
pylab.plot(x,y2,'-b',label='cos(x)')
pylab.xlabel('x (radians)')
pylab.ylabel('y')
pylab.legend()
pylab.savefig('fig2.eps')
```

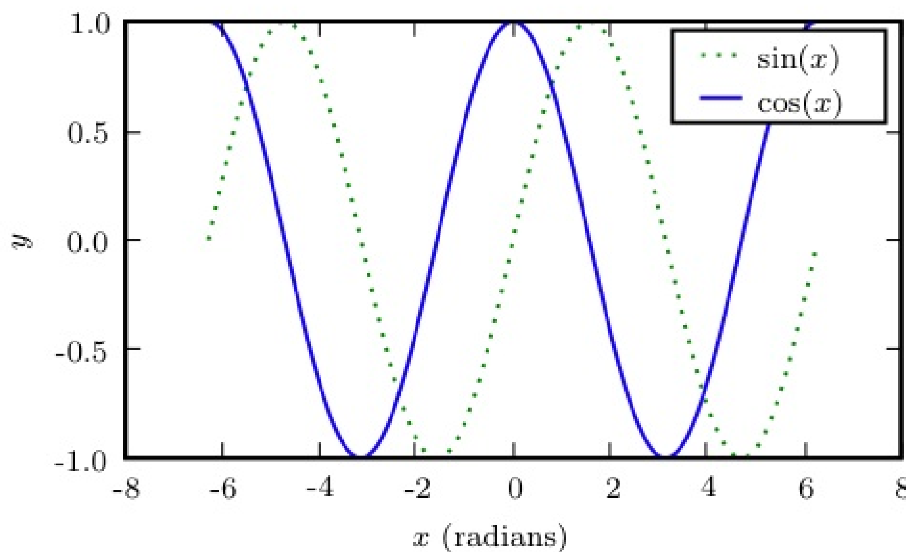


FIG. 1: Plot of the sine and cosine functions.

Odds and Ends

Another way to set the legend fonts after the legend has been drawn is to use, for example:

```
from matplotlib.font_manager import fontManager, FontProperties
font= FontProperties(size='x-small');
pylab.legend(loc=0, prop=font);
```

– DavidDonovan

PDF File Size

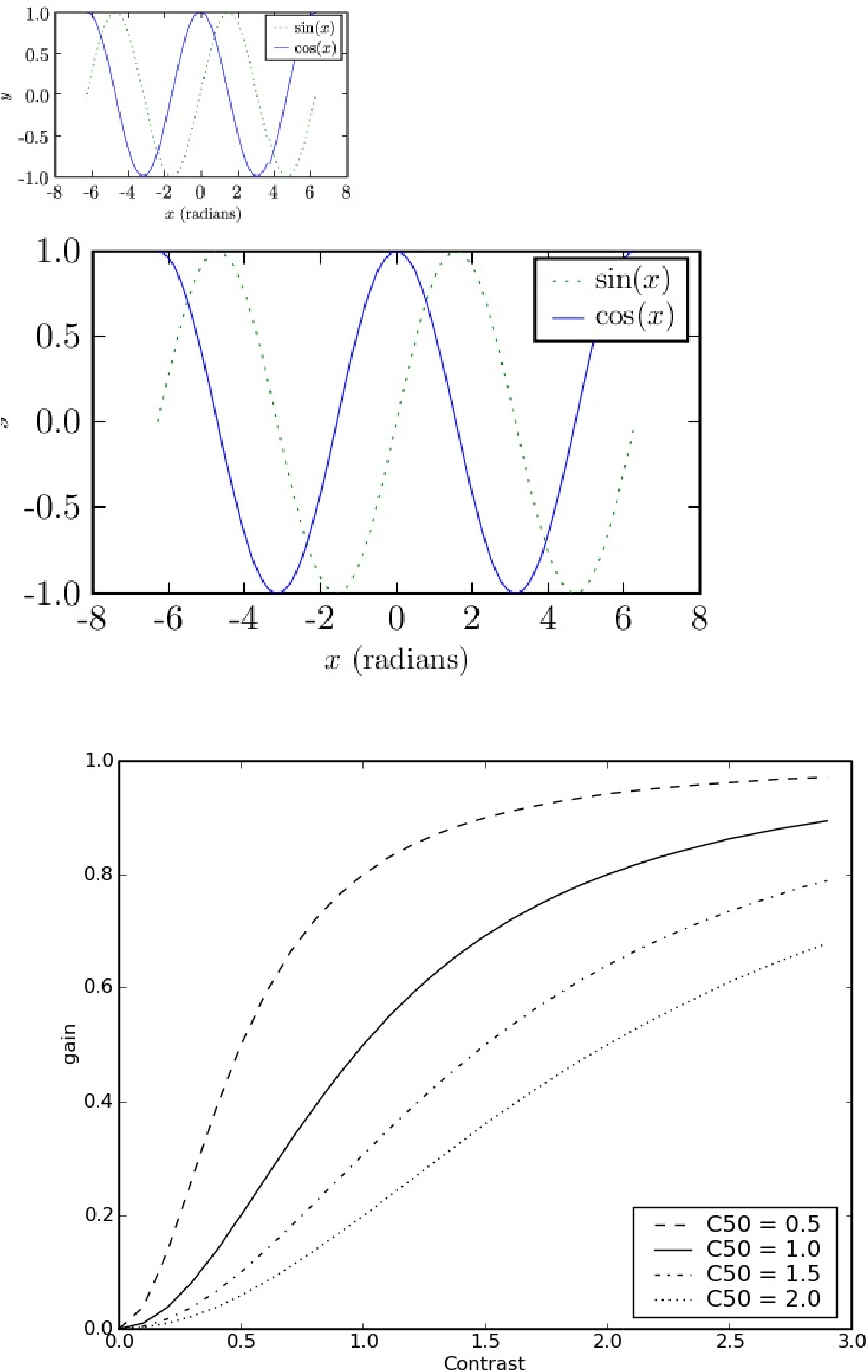
If you have very complex images such as high-resolution contour plots or three-dimensional plots, saving these to vector graphic formats like PDF or EPS can result in unacceptably large files (though with the ability for stunning zooms). One solution is to selectively convert parts of the plot (not the text labels) to a rasterized image. This can be done with the so-called “mixed-mode rendering” capability in recent versions of matplotlib. Here is an example as it should probably work:

```
#!python
from pylab import meshgrid, sin, cos, linspace, contourf, savefig,
x, y = meshgrid(*(linspace(-1,1,500),)*2)
z = sin(20*x**2)*cos(30*y)
c = contourf(x,y,z,50)
savefig('full_vector.pdf')
clf()
c = contourf(x,y,z,50,rasterized=True)
savefig('rasterized.pdf')
```

however, {{{contourf}}} currently does not support the {{{rasterized}}} option (which is silently ignored). Some other plot elements do however. I am looking into a solution for this. – MichaelMcNeilForbes

Attachments

- `fig.png`
- `fig1.png`
- `naka-rushton.png`
- `psfrag_example.png`



sin(x)
cos(x)
y
x (radians)
sin(x)
cos(x)
y
x (radians)
C50 = 0.5
C50 = 1.0
C50 = 1.5
C50 = 2.0
gain
Contrast

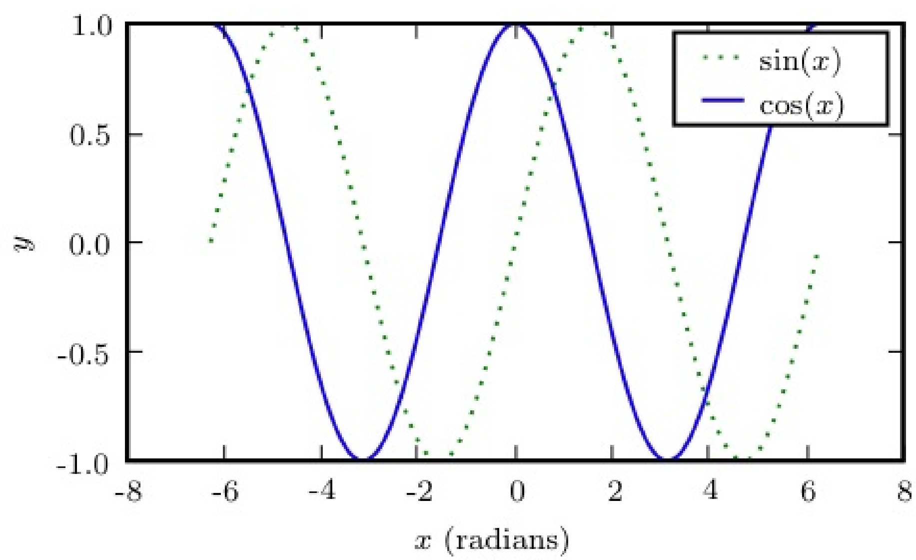


FIG. 1: Plot of the sine and cosine functions.

Matplotlib: using tex

Matplotlib can use LaTeX to handle the text layout in your figures. This option (which is still somewhat experimental) can be activated by setting text.usetex : true in your rc settings. Text handling with matplotlib's LaTeX support is slower than standard text handling, but is more flexible, and produces publication-quality plots. The results are striking, especially when you take care to use the same fonts in your figures as in the main document.

Matplotlib's LaTeX support is still under development, although at least two individuals have relied upon it to generate the figures for their doctoral dissertations. Many improvements have been made beginning with matplotlib-0.87, please update matplotlib if you have an earlier version. This option requires a working LaTeX installation, dvipng (which may be included with your TeX installation), and ghostscript (AFPL, GPL, or ESP ghostscript should all work, but GPL ghostscript-8.60 or later is recommended). The executables for these external dependencies must be located on your PATH.

There are a couple of options to mention, which can be changed using rc settings, either using a matplotlibrc file, or the rcParams dict in your program. Here is an example matplotlibrc file:

```
font.family         : serif
font.serif          : Times, Palatino, New Century Schoolbook, Bookr
font.sans-serif     : Helvetica, Avant Garde, Computer Modern Sans s
font.cursive        : Zapf Chancery
font.monospace      : Courier, Computer Modern Typewriter

text.usetex         : true
```

The first valid font in each family is the one that will be loaded. If the fonts are not specified, the Computer Modern fonts are used by default. All of the other fonts are Adobe fonts. Times and Palatino each have their own accompanying math fonts, while the other Adobe serif fonts make use of the Computer Modern math fonts. See psnfss2e.pdf for more details.

To use tex and select e.g. Helvetica as the default font, without edititing matplotlibrc use:

```
#!python
from matplotlib import rc
rc('font',**{'family':'sans-serif','sans-serif':['Helvetica']})
## for Palatino and other serif fonts use:
#rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)
```

N.B. You need to do this *before* you import `matplotlib.pylab`.

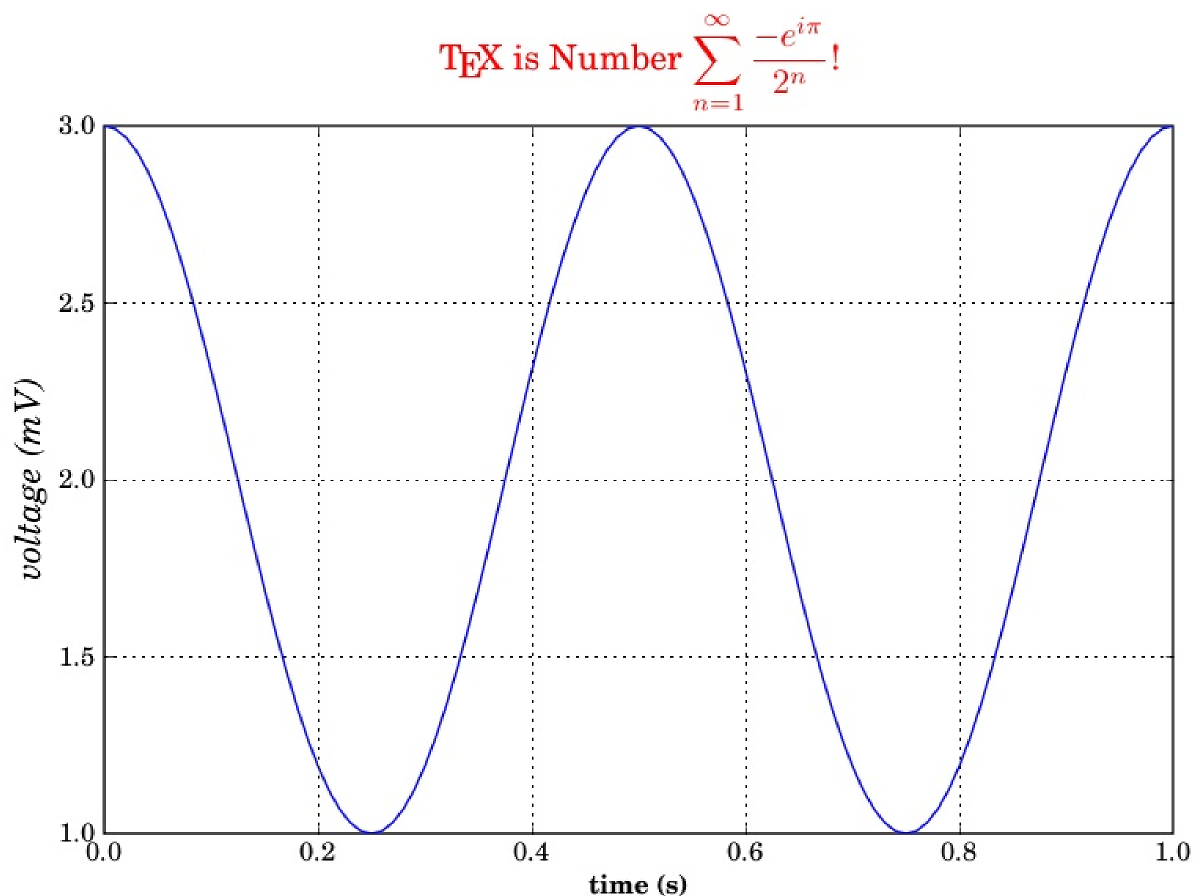
Here is the standard example, tex_demo.py:

```
from matplotlib import rc
from matplotlib.numerix import arange, cos, pi
from pylab import figure, axes, plot, xlabel, ylabel, title, grid,

rc('text', usetex=True)
figure(1)
ax = axes([0.1, 0.1, 0.8, 0.7])
t = arange(0.0, 1.0+0.01, 0.01)
s = cos(2*2*pi*t)+2
plot(t, s)

xlabel(r'\textbf{time (s)}')
ylabel(r'\textit{voltage (mV)}',fontsize=16)
title(r"\TeX\ is Number $\displaystyle\sum_{n=1}^\infty\frac{-e^{i\
grid(True)
savefig('tex_demo')

show()
```



Note that when TeX/LaTeX support is enabled, you can mix text and math modes. Display math mode ($$ e=mc^2 $$) is not supported, but adding the command , as in tex_demo.py, will produce the same results.

In order to produce encapsulated postscript files that can be embedded in a new LaTeX document, the default behavior of matplotlib is to distill the output, which removes some postscript operators used by LaTeX that are illegal in an eps file. This step produces fonts which may be unacceptable to some users. One workaround is to to set ps.distiller.res to a higher value (perhaps 6000) in your rc settings. A better workaround, which requires [http://www.foolabs.com/xpdf/download.html xpdf] or [http://poppler.freedesktop.org/ poppler] (the new backend to xpdf) can be activated by changing the rc ps.usedistiller setting to xpdf. The xpdf alternative produces postscript with text that can be edited in Adobe Illustrator, or searched for once converted to pdf.

Possible Hangups

- On Windows, the PATH environment variable may need to be modified to find the latex, dvipng and ghostscript executables. This is done by going to the control panel, selecting the “system” icon, selecting the “advanced” tab, and clicking the “environment variables” button (and people think Linux is complicated. Sheesh.) Select the PATH variable, and add the appropriate

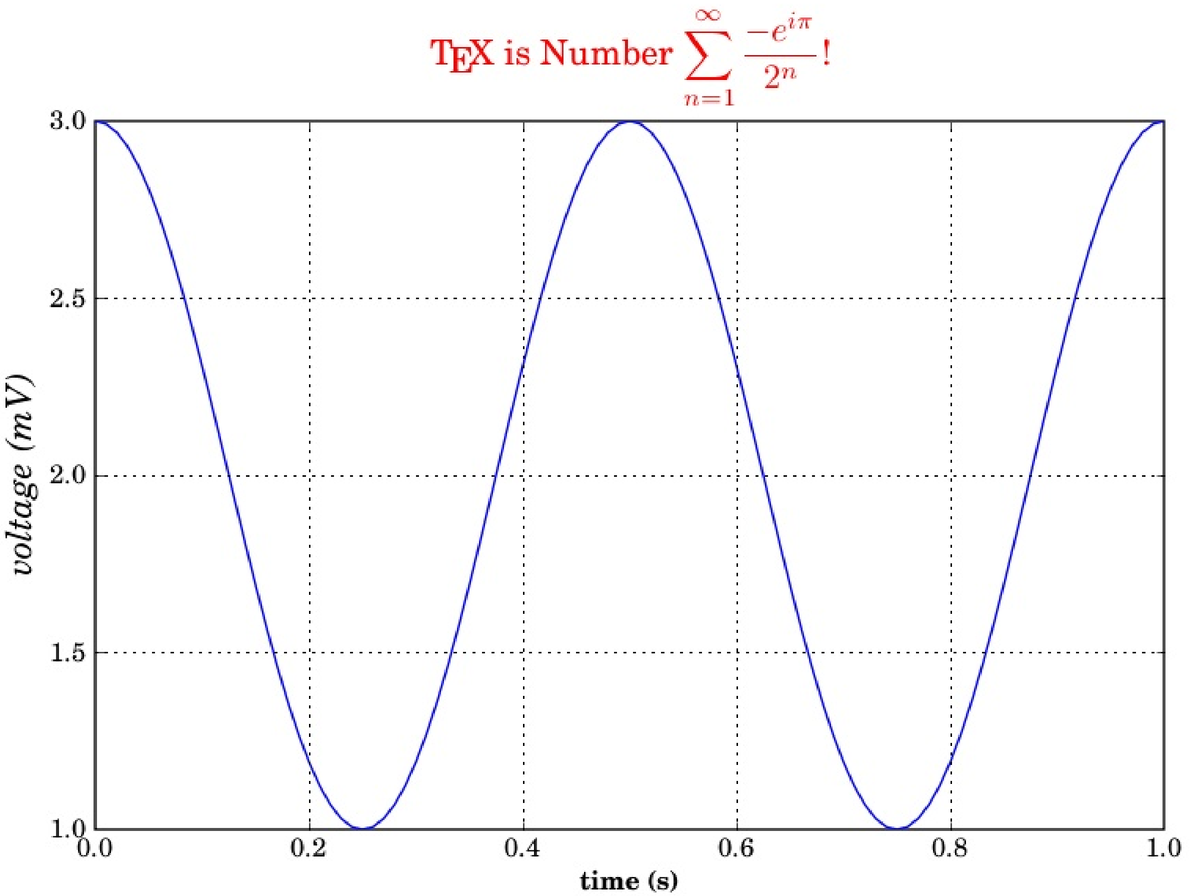
directories.
- Using MiKTeX with Computer Modern fonts, if you get odd -Agg and PNG results, go to MiKTeX/Options and update your format files
- The fonts look terrible on screen. You are probably running Mac OS, and there is some funny business with dvipng on the mac. Set text.dvipnghack : True in your matplotlibrc file.
- On Ubuntu and Gentoo, the base texlive install does not ship with the type1cm package. You may need to install some of the extra packages to get all the goodies that come bundled with other latex distributions.
- Some progress has been made so Matplotlib uses the dvi files directly for text layout. This allows latex to be used for text layout with the pdf and svg backends, as well as the *Agg and PS backends. In the future, a latex installation may be the only external dependency.

In the event that things dont work

- Try `rm -r ~/.matplotlib/*cache`
- Make sure LaTeX, dvipng and ghostscript are each working and on your PATH.
- Run your script with verbose mode enabled: python example.py –verbose-helpful (or –verbose-debug-annoying) and inspect the output. Most likely there is a problem reported there. If you need help, post a short example that reproduces the behavior, explicitly noting any changes to your rc settings, along with what version of matplotlib you have installed, your os, and the –verbose-* output.

Attachments

- `tex_demo.png`


TEX is Number $\sum_{n=1}^{\infty} \frac{-e^{i\pi}}{2^n}$!
3.0
2.5
2.0
1.5
1.0
voltage (mV)
0.0
0.2
0.4
0.6
0.8
1.0
time (s)

Mayavi

Mayavi surf

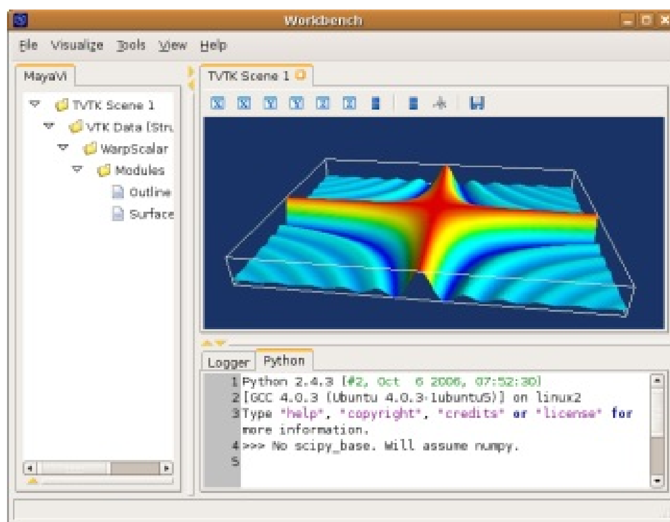
If you want to plot a surface representing a matrix by elevation and colour of its points you have to transform the matrix data in a 3D data that !MayaVi2 can understand. [:Cookbook/MayaVi/mlab:mlab] knows how to do this, but it does not have the nice user interface of !MayaVi2. Here is a script that create a !SurfRegular object using mlab, and then loads it in !MayaVi2. A more detailed version of this script is given in the examples pages [:Cookbook/MayaVi/Examples].

```
import numpy
def f(x, y):
    return numpy.sin(x*y)/(x*y)
x = numpy.arange(-7., 7.05, 0.1)
y = numpy.arange(-5., 5.05, 0.05)
from enthought.tvtk.tools import mlab
s = mlab.SurfRegular(x, y, f)
from enthought.mayavi.sources.vtk_data_source import VTKDataSource
d = VTKDataSource()
d.data = s.data
mayavi.add_source(d)
from enthought.mayavi.filters.warp_scalar import WarpScalar
w = WarpScalar()
mayavi.add_filter(w)
from enthought.mayavi.modules.outline import Outline
from enthought.mayavi.modules.surface import Surface
o = Outline()
s = Surface()
mayavi.add_module(o)
mayavi.add_module(s)
```

You can run this script by running “mayavi2 -n -x script.py”, loading it through the menu (File -> Open File), and pressing Ctrl+R, or entering “execfile(‘script.py’) in the python shell.

Attachments

- `surf.png`


Workbench
File Visualize Tools View Help
MayaVi
TVTK Scene 1
WarpScalar
Modules
Outline
Surface
TVTK Scene 1
Logger
Python
1 Python 2.4.3 (#2, Oct 6 2006, 07:52:30)
2 [GCC 4.0.3 (Ubuntu 4.0.3-1ubuntu5)] on linux2
3 Type "help", "copyright", "credits" or "license" for
more information.
4 >>> No scipy_base. Will assume numpy.
5

Mayavi tips

Introduction

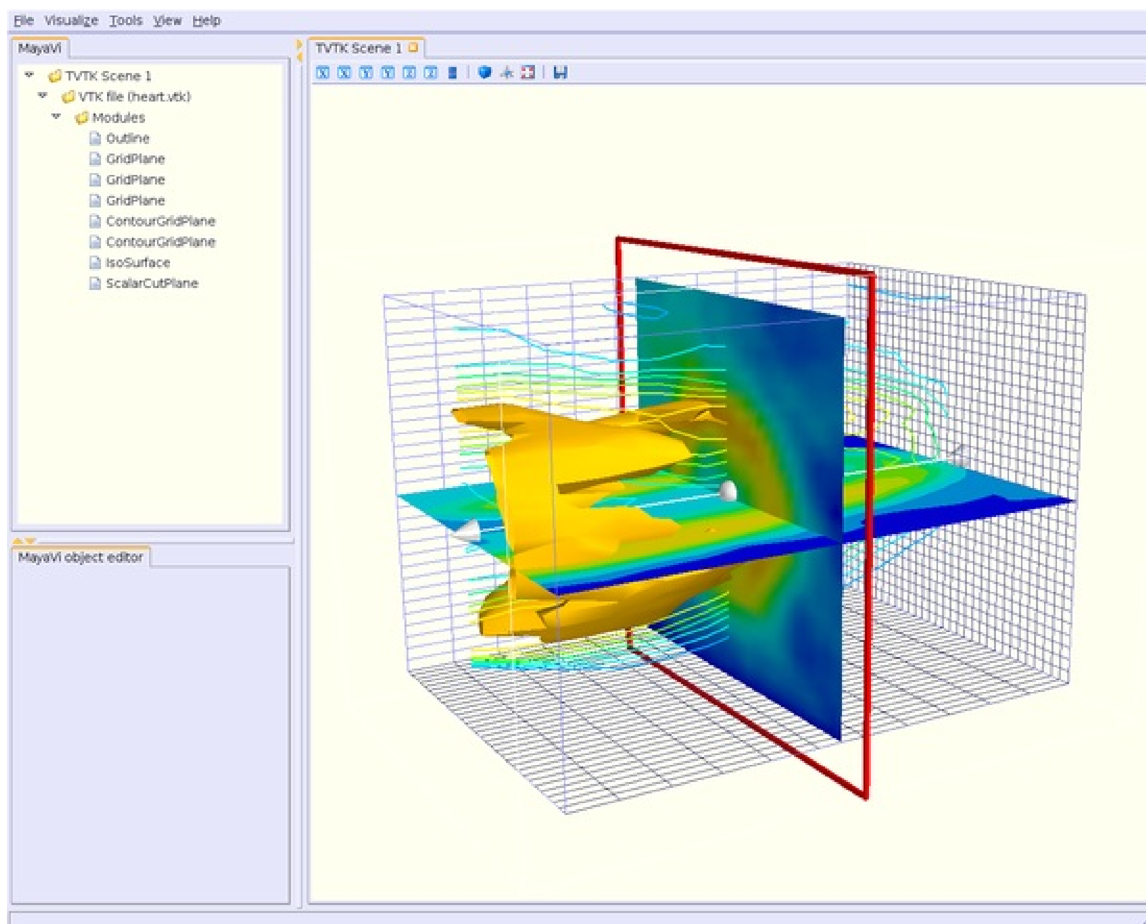
Here are general tips on how best to use !MayaVi2.

Scripting MayaVi2 effectively

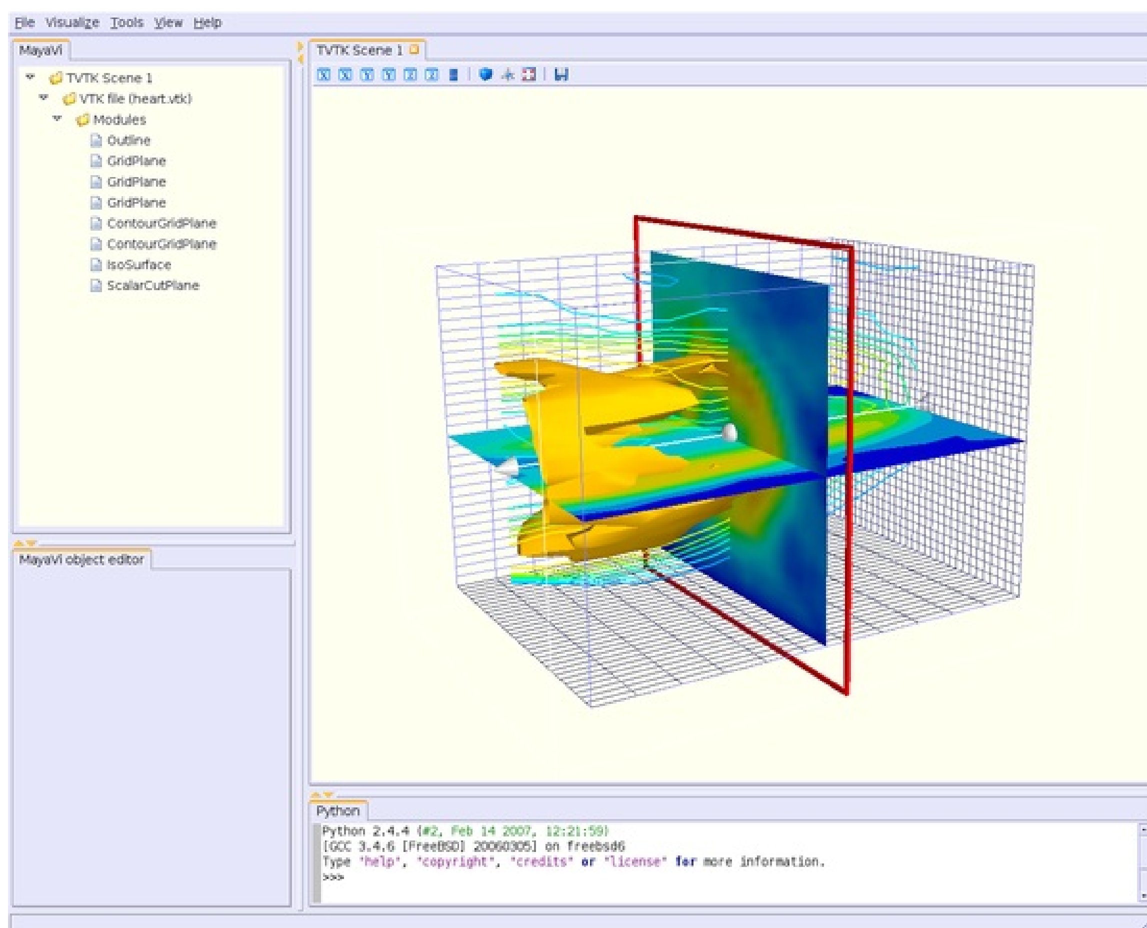
Here are a few tips showing how to script mayavi2 interactively and effectively.

Drag and drop object

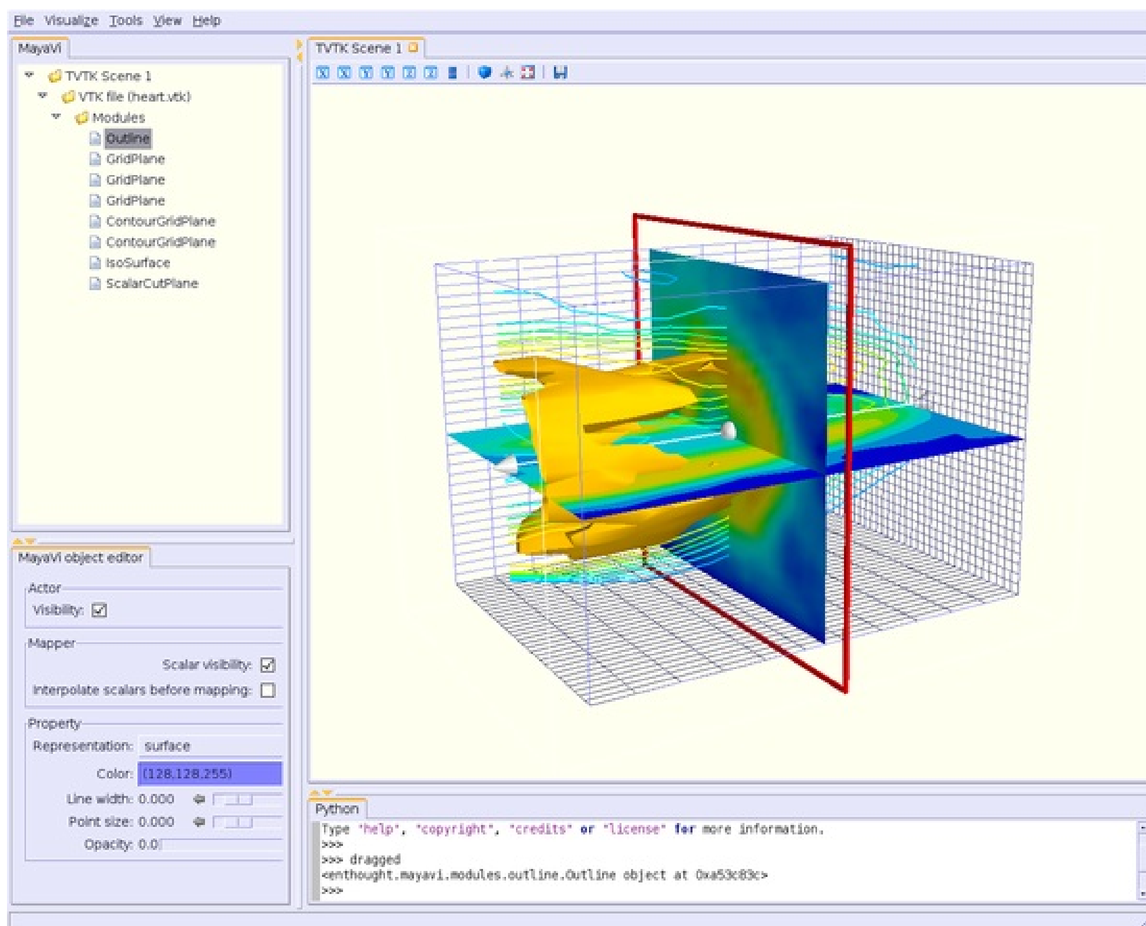
Running the contour.py python script example, you should get:



First, enable python shell clicking on Python in the “View” menu. A python shell should appear at the bottom of the !MayaVi2 window:



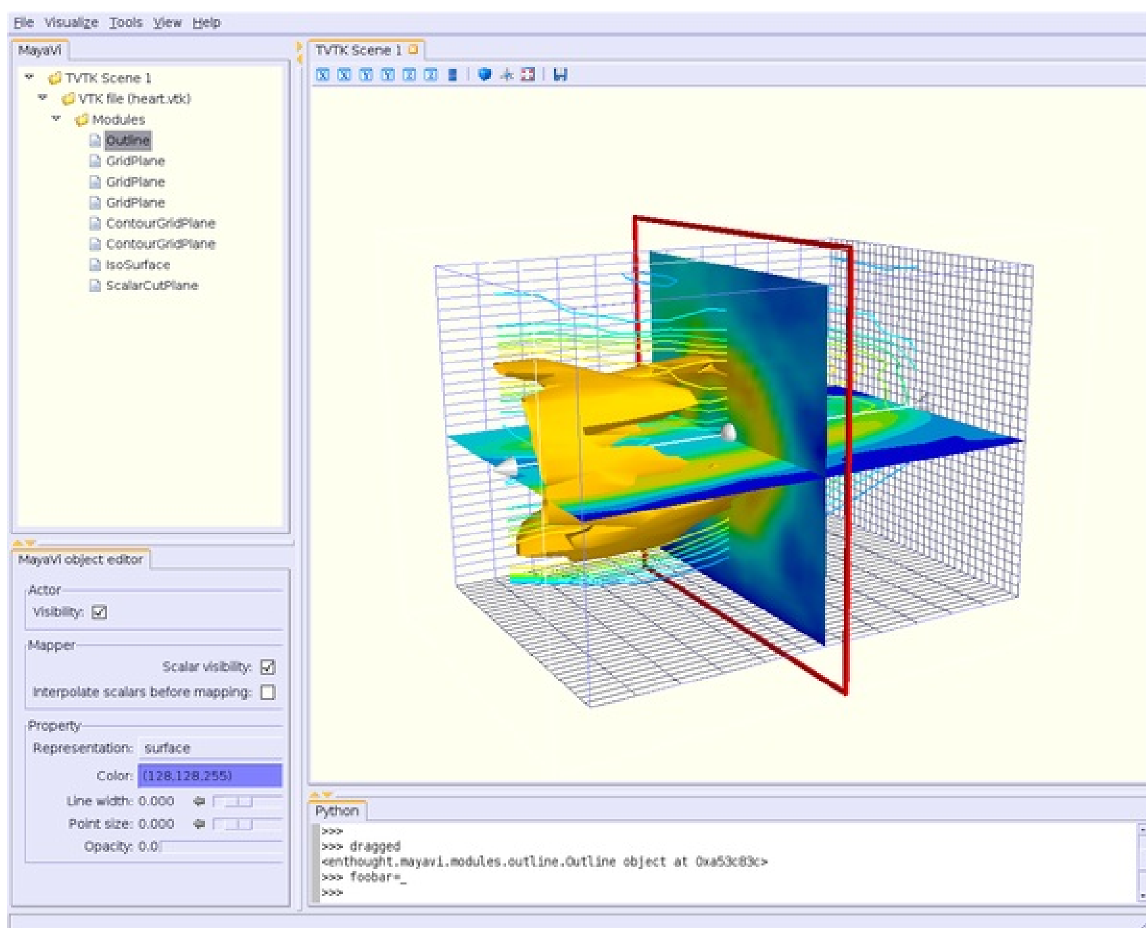
Then drag any object from the tree view on the left and drop it on the python shell and you'll get the object. Say you want to get the Outline module:



Now, you can use your object following two ways: typing directly in the python shell or using the explore() method.

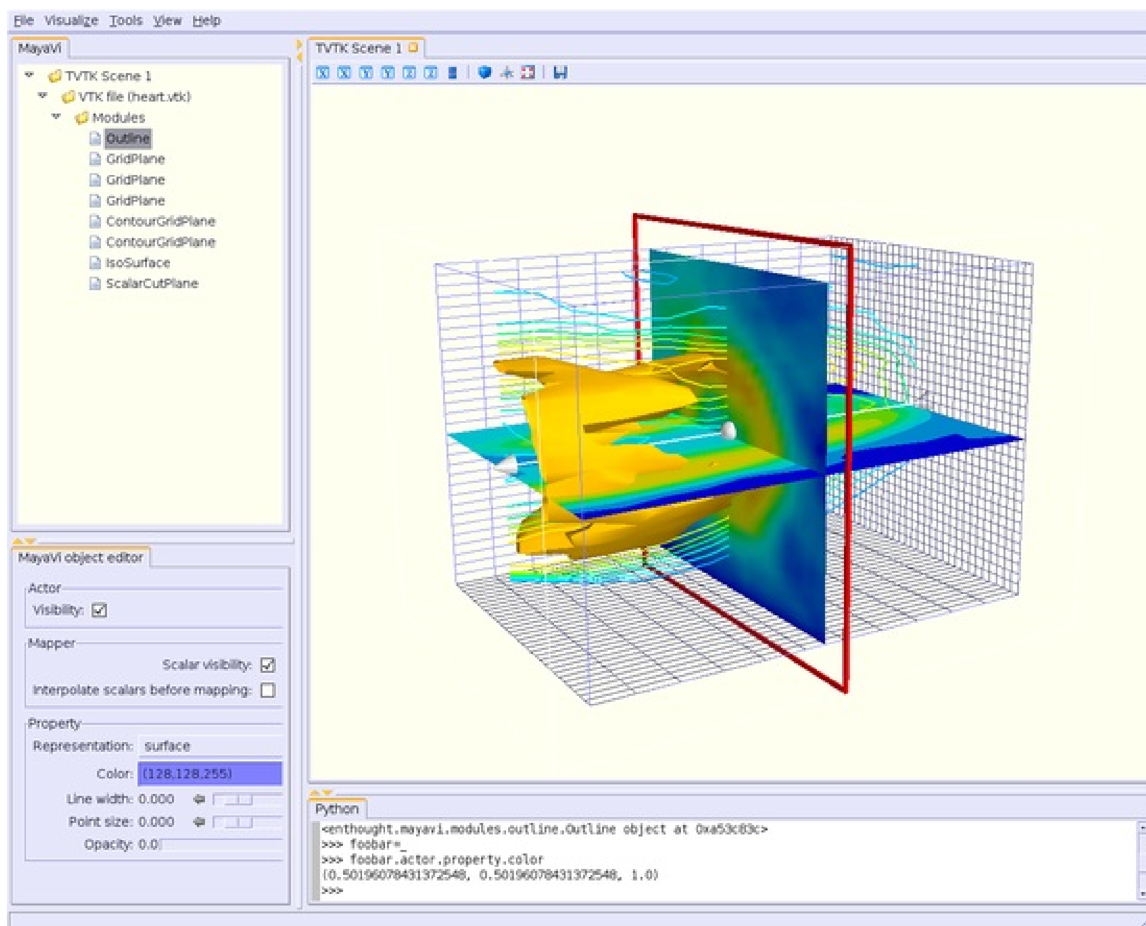
Typing in the python shell

You can create an instance of your object in the python shell window.



Note that you benefit of “word completion” in this python shell, ie a little window popups letting you choose the name of the expected object or method.

Thus, you can display the RGB values of the outline color for instance:



However, find out objects or methods can be not so easy: you may not know how they depends from each other. An easier way is using the explore() method.

Using the explore() method

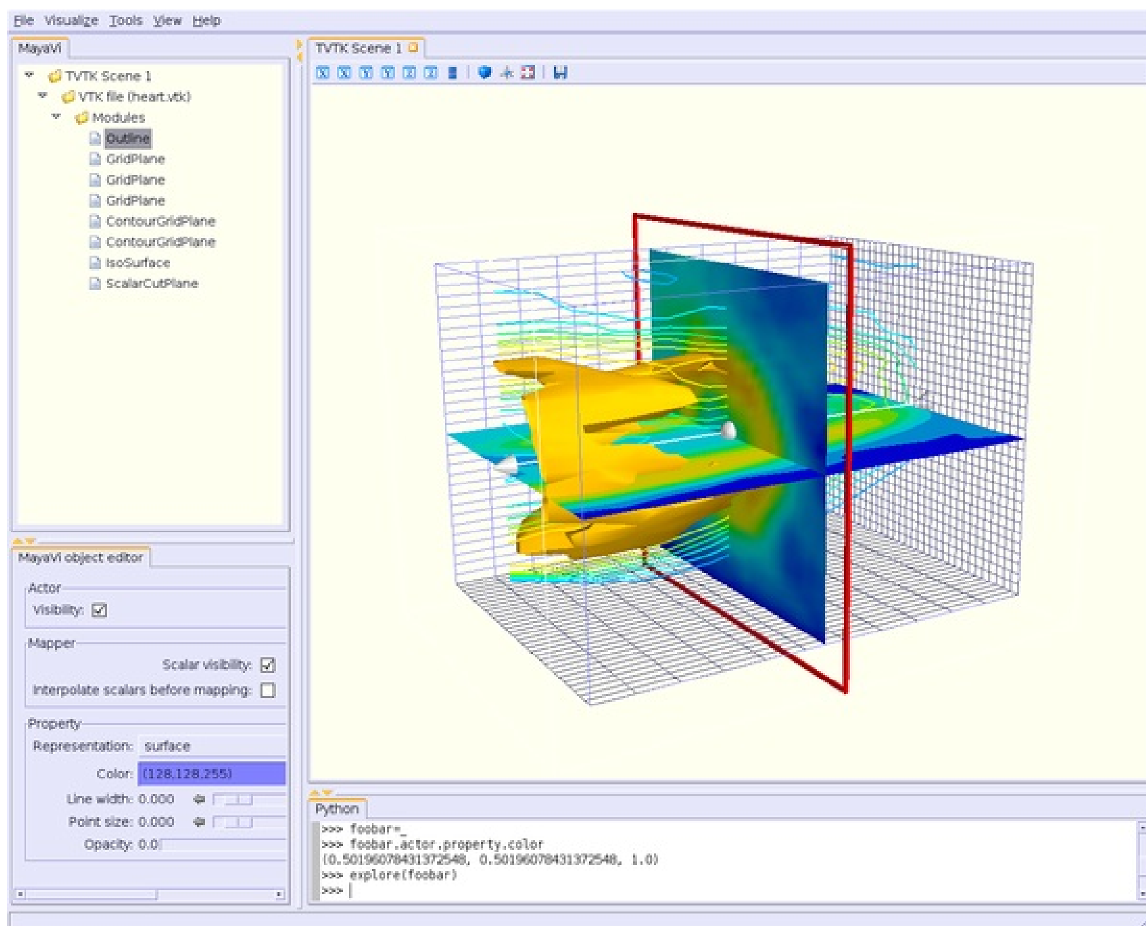
Simply type

```
explore(_)
```

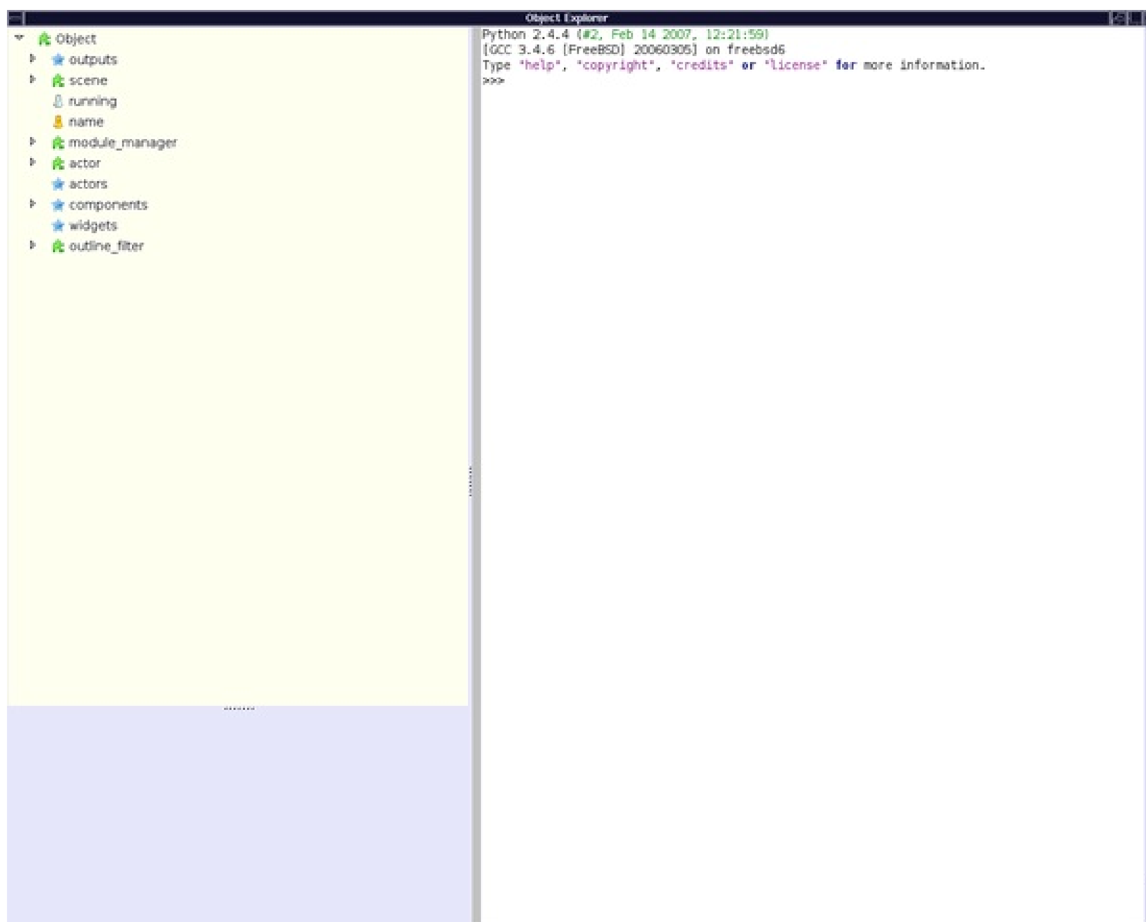
or

```
explore(foobar)
```

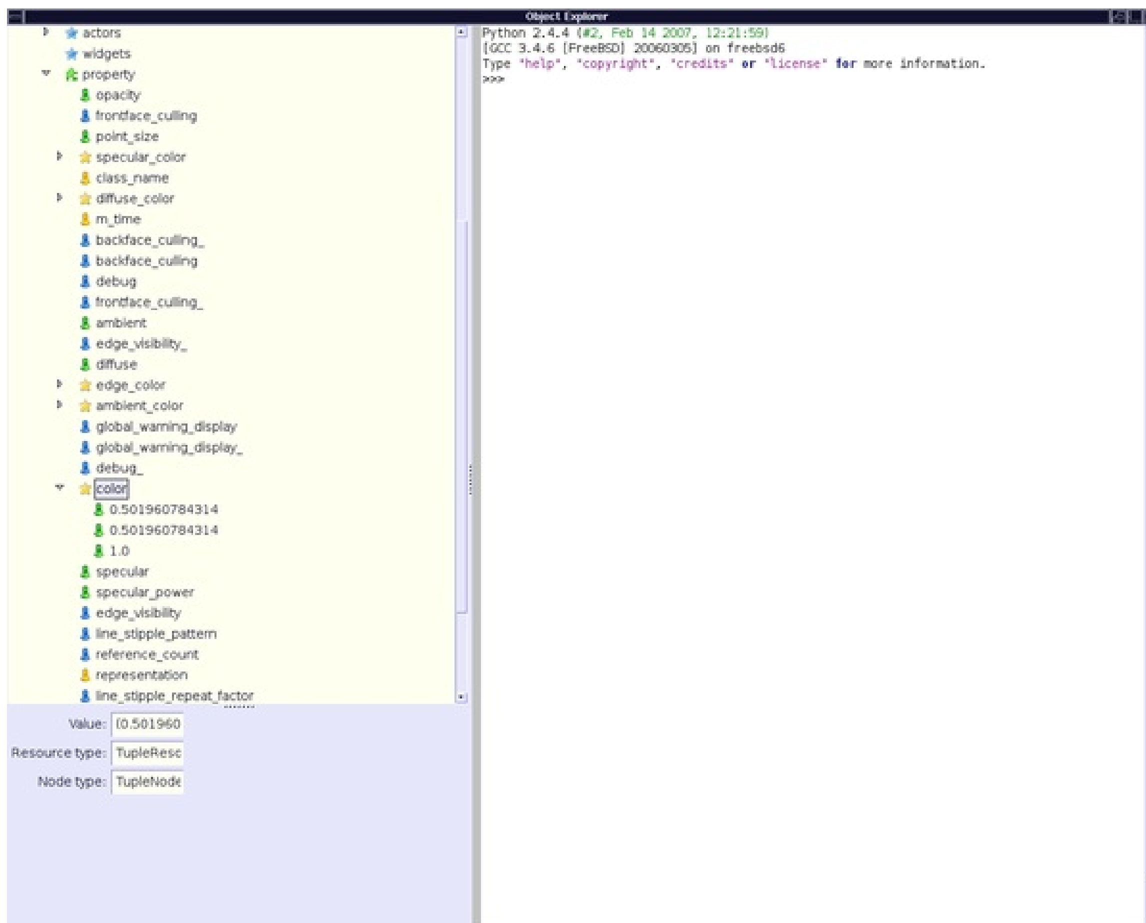
if you have previously defined it:



Then, you get the following window:



Considering the last example, about the color of the outline module, you can unfold the “tree” and thus, get the info you need:



Very powerful, isn’t it ! :-)

You can also work in the pyhon shell, create your objects and so on...

Working in the python shell

On the embedded Python shell in the !MayaVi2 application, the name ‘mayavi’ is bound to the !MayaVi scripting interface which can be used to script mayavi easily as shown here:

```
e = mayavi.engine # Get the MayaVi engine.
mayavi.new_scene() # Create a new scene
# [...]
mayavi.save_visualization('foo.mv2')
mayavi.load_visualization('foo.mv2')
```

Note that the Mayavi engine lets you script mayavi in powerful ways. For example:

```
e = mayavi.engine
```

Here ‘e’ is the running Engine instance (mayavi/engine.py) and has the same hierarchy that you see on the mayavi tree view. For example:

```
e.scenes[0]             # --> first scene in mayavi.
e.scenes[0].children[0] # --> first scene's first source (vtkfile)
```

Another example, just run examples/contour.py and type/cut/paste the following to get the scalar cut plane module:

```
e = mayavi.engine
s = e.scenes[0].children[0].children[0].children[-1]
# Think scene.datafile.module_manager.last_module
```

It is possible to use Mayavi2 from within IPython and script it. You must first start IPython with the ‘-wthread’ command line option. Here is an example:

```
from enthought.mayavi.app import Mayavi
m = Mayavi()
m.main()
m.script.new_scene() # m.script is the mayavi.script.Script instanc
engine = m.script.engine
```

In the above, ‘m.script’ is the same as the ‘mayavi’ instance on the embedded shell.

If you are writing a stand-alone script to visualize something like the scripts in the examples/ directory, please use any of the examples as a template.

Save snapshots

Saving snapshots within a python script is very easy:

```
s = script.engine.current_scene
s.scene.save('test.jpg', size=(width,height))
```

You can also save images in a lot of others format: !PostScript (ps), Encapsuled !PostScript (eps), PDF (pdf), Bitmap (bmp), TIFF (tiff), PNG (png), !OpenInventor (iv), Virtual Reality Markup Language (wrl, vrml), Geomview (oogl), !RenderMan RIB (rib), Wavefront (obj).

The obvious corollary of saving snapshots is saving a lot of snapshots in order to make a movie for example, without !MayaVi2 window popup for each snapshot recorded.

The answer is straightforward (only under UN*X boxes): use the ‘X virtual framebuffer’.

The following lines give you the trick. You can improve it, of course, scripting it in shell, python, and so on.

```
* create your X virtual framebuffer with the following command: 'xv
```

```
* export your display: 'export DISPLAY=:1' (sh/bash syntax) or 'sete
```

```
* run your !MayaVi2 script as usual;
```

```
* once finished, and all your snapshots have been created, don't fo
```

Enabling alpha transparency in the colormap

Drag the module manager to the python shell and you will be able to enable alpha transparency in the colormap via:

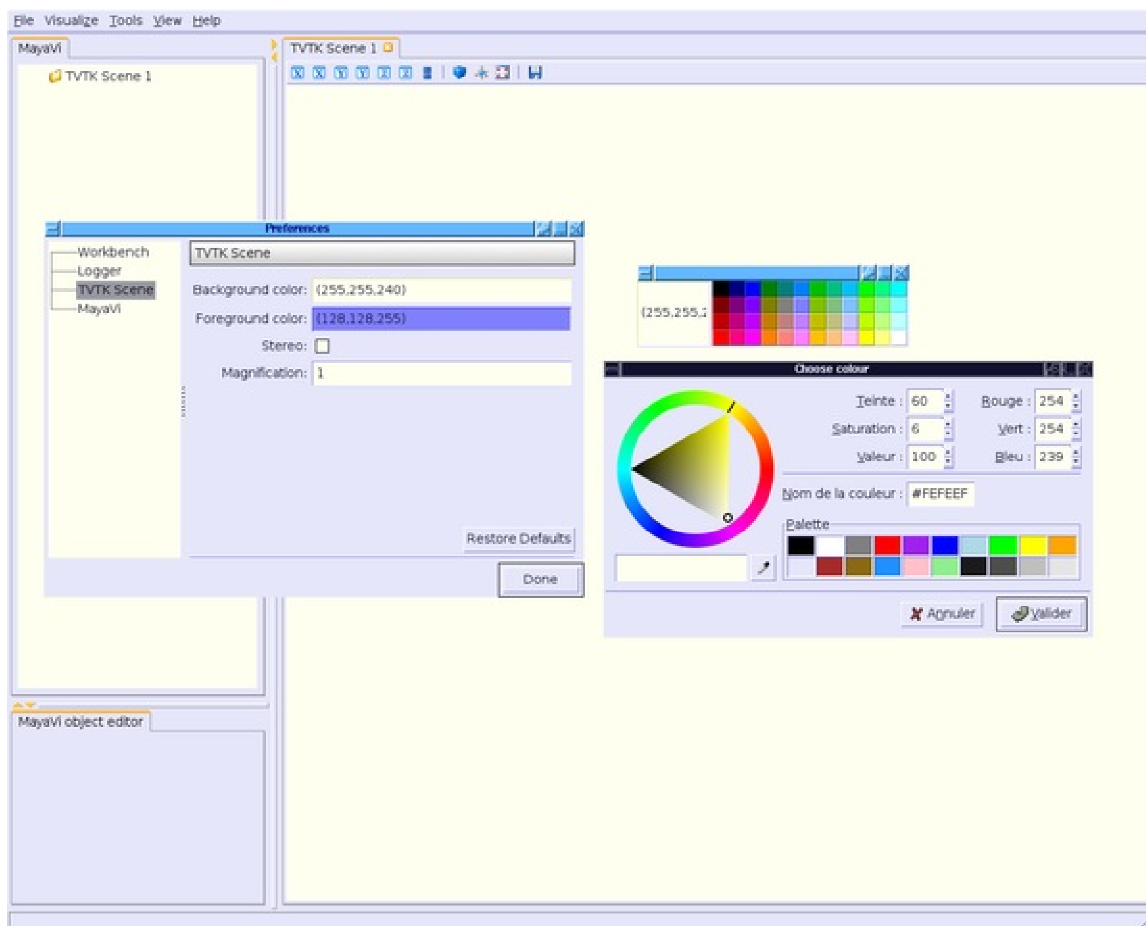
```
dragged.scalar_lut_manager.lut.alpha_range=(0,1)
```

Set MayaVi2 session colors

Run !MayaVi2, go to “Tools” menu then “Preferences” then “TVTK Scene”.

Suppose that you want to change background color: click on “Background color” tag.

Here, you can choose a predefined color, or click in the square to set your RGB value, for instance.



Also, if you want to set foreground color, it will be applied for all modules and filters, i.e. outline color, text color, labels axes, and so on.

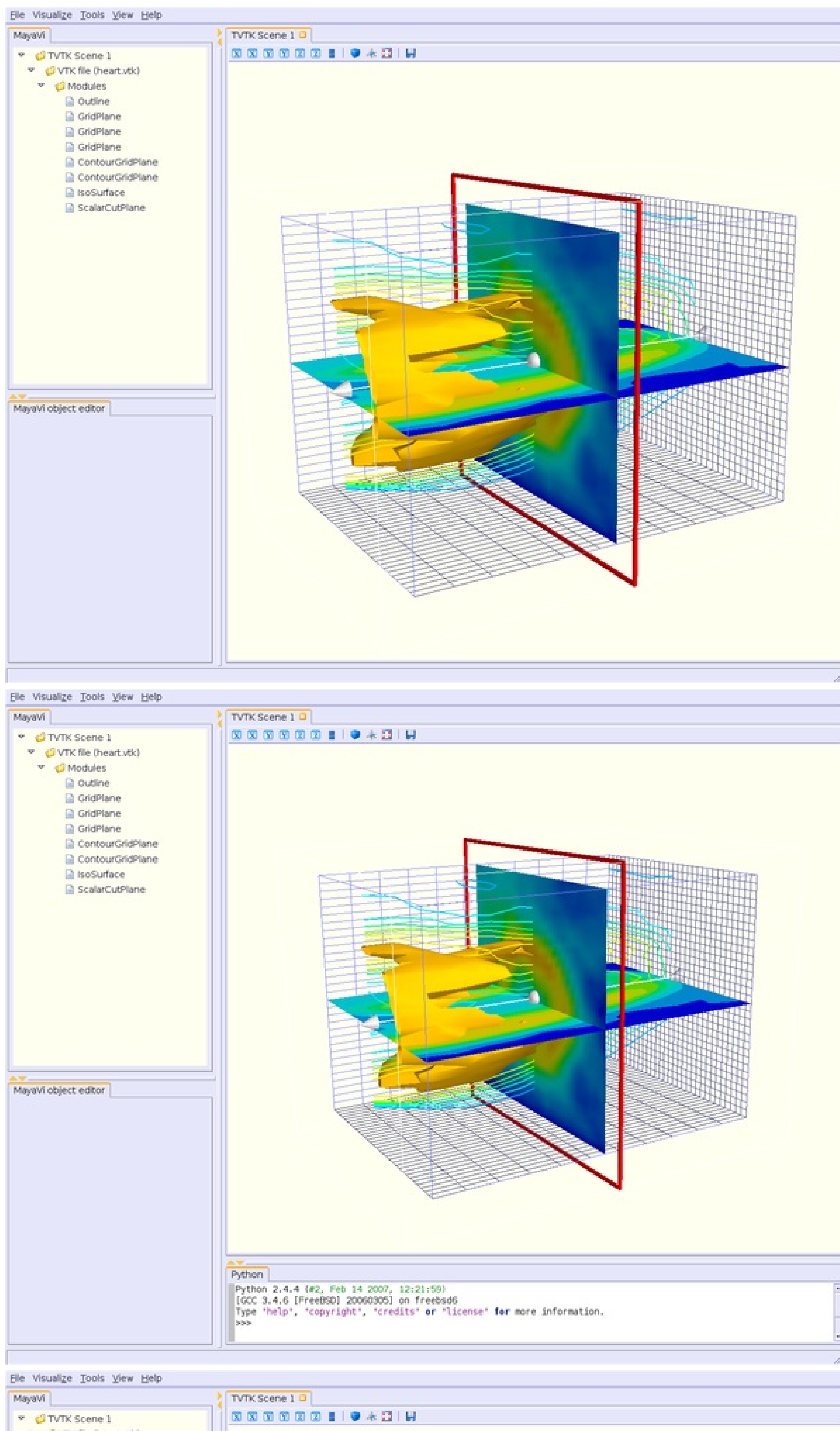
Your preferences will be saved in a !MayaVi2 configuration file, so you'll get these colors each time you run a !MayaVi2 session.

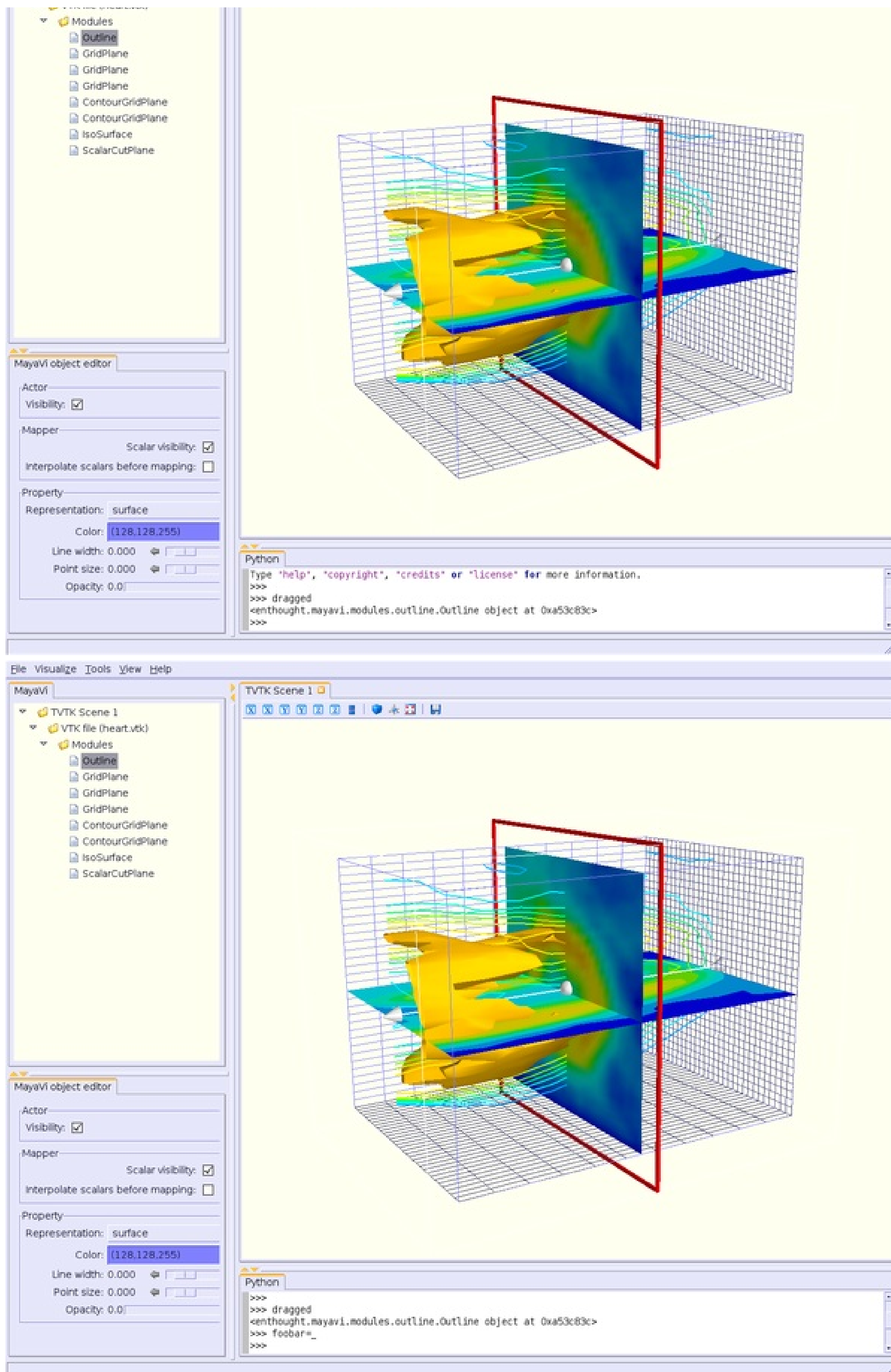
= Writing VTK data files using TVTK =

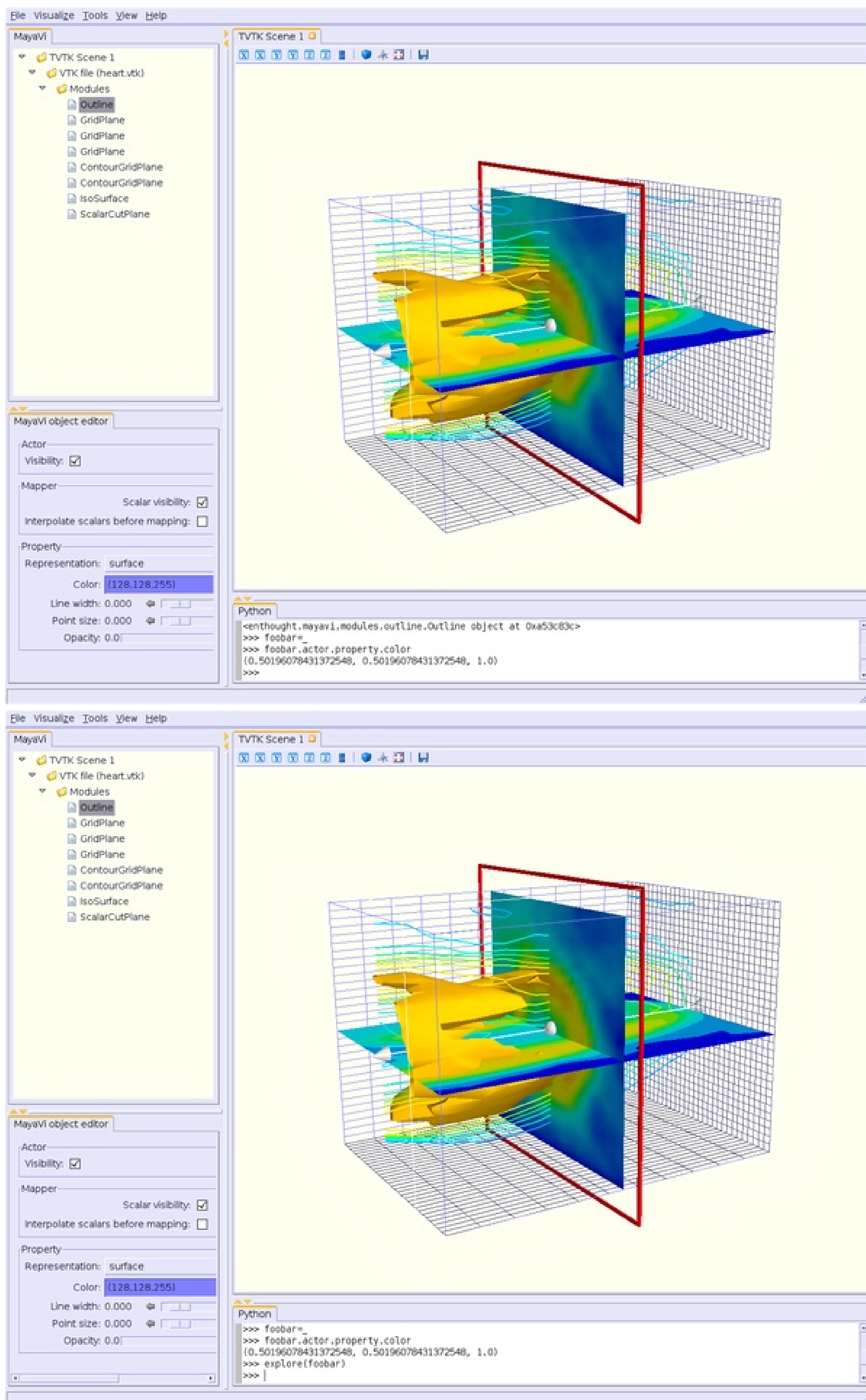
Coming soon...

Attachments

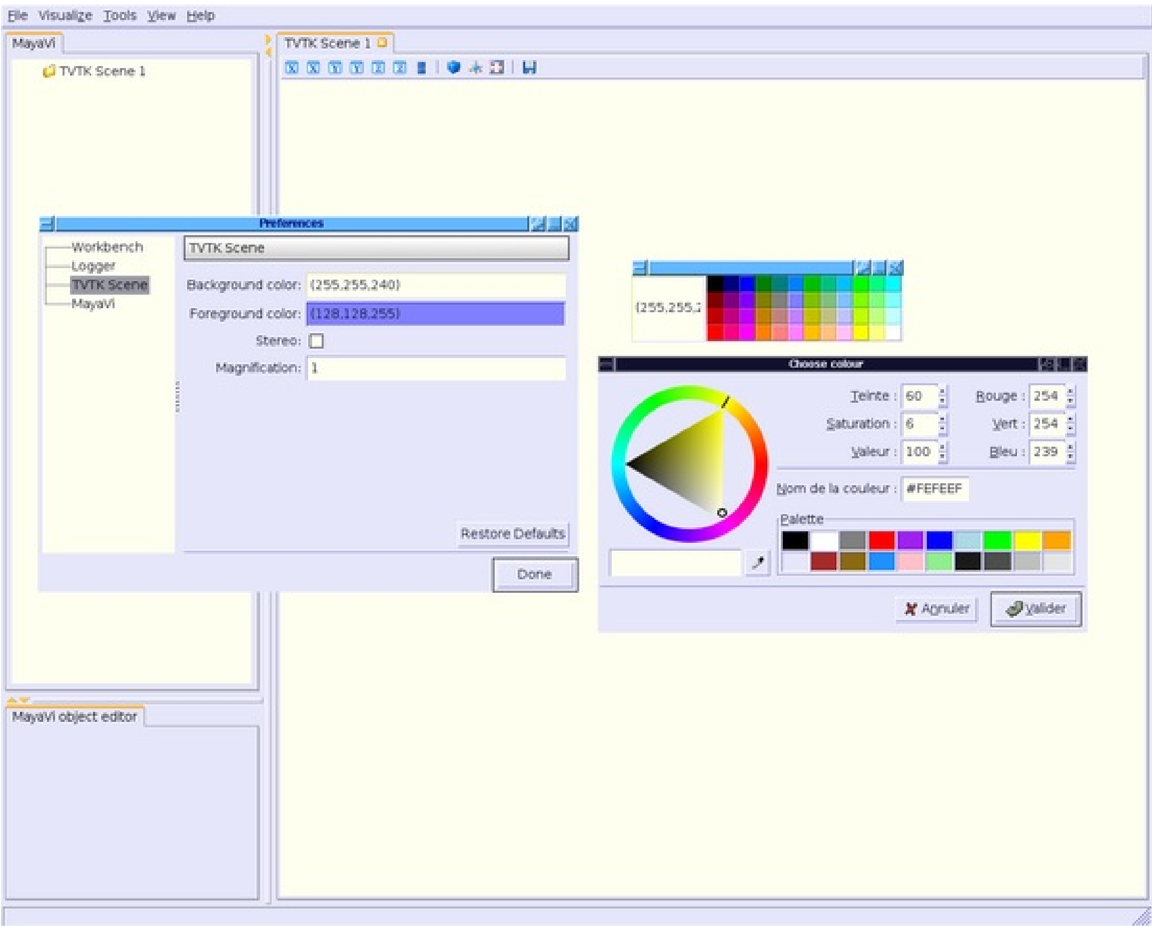
- `ps1.png`
- `ps2.png`
- `ps3.png`
- `ps4.png`
- `ps5.png`
- `ps6.png`
- `ps7.png`
- `ps8.png`
- `setcolors.png`


File Visualize Tools View Help
MayaVi
TVTK Scene 1
VTK file (heart.vtk)
Modules
Outline
GridPlane
GridPlane
GridPlane
ContourGridPlane
ContourGridPlane
IsoSurface
ScalarCutPlane
MayaVi object editor
Python
Python 2.4.4 (#2, Feb 14 2007, 12:21:59)
[GCC 3.4.6 [FreeBSD] 20060305] on freebsd6
Type "help", "copyright", "credits" or "license" for more information.
>>>




File Visualize Tools View Help
MayaVi
TVTK Scene 1
VTK file (heart.vtk)
Modules
Outline
GridPlane
GridPlane
GridPlane
ContourGridPlane
ContourGridPlane
IsoSurface
ScalarCutPlane
MayaVi object editor
Actor
Visibility:
Mapper
Scalar visibility:
Interpolate scalars before mapping:
Property
Representation: surface
Color: (128,128,255)
Line width: 0.000
Point size: 0.000
Opacity: 0.0
Python
<enthought.mayavi.modules.outline.Outline object at 0xa53c83c>
>>> foobar=_
>>> foobar.actor.property.color
(0.50196078431372548, 0.50196078431372548, 1.0)
>>>
>>> foobar=_
>>> foobar.actor.property.color
(0.50196078431372548, 0.50196078431372548, 1.0)
>>> explore(foobar)
>>>


Object Explorer
Object
outputs
scene
running
name
module_manager
actor
actors
components
widgets
outline_filter
Python 2.4.4 (#2, Feb 14 2007, 12:21:59)
[GCC 3.4.6 [FreeBSD] 20060305] on freebsd6
Type "help", "copyright", "credits" or "license" for more information.
>>>
Object Explorer
actors
widgets
property
opacity
frontface_culling
point_size
specular_color
class_name
diffuse_color
m_time
backface_culling_
backface_culling
debug
frontface_culling_
ambient
edge_visibility_
diffuse
edge_color
ambient_color
global_warning_display
global_warning_display_
debug_
color
0.501960784314
0.501960784314
1.0
specular
specular_power
edge_visibility
line_stipple_pattern
reference_count
representation
line_stipple_repeat_factor
Value:
Resource type:
Node type:
Python 2.4.4 (#2, Feb 14 2007, 12:21:59)
[GCC 3.4.6 [FreeBSD] 20060305] on freebsd6
Type "help", "copyright", "credits" or "license" for more information.
>>>


File Visualize Tools View Help
MayaVi
TVTK Scene 1
TVTK Scene 1
Preferences
Workbench
Logger
TVTK Scene
MayaVi
TVTK Scene
Background color: (255,255,240)
Foreground color: (128,128,255)
Stereo:
Magnification: 1
Restore Defaults
Done
Choose colour
Teinte : 60
Saturation : 6
Valeur : 100
Rouge : 254
Vert : 254
Bleu : 239
Nom de la couleur : #FEFEEF
Palette
Annuler
Valider
MayaVi object editor

Mayavi: running mayavi 2

Introduction

!MayaVi2 works with two kinds of “objects”: modules and filters.

!MayaVi Users Guide (v. 1.5) by P. Ramachandran defines them as follows:

```
* "A Module is an object that actually visualizes the data.";
```

```
* "A Filter is an object that filters out the data in some way or th
```

You can run !MayaVi2 using one of the following two ways:

```
* run !MayaVi2 as is, without any options on the command line. You'l
```

```
* run !MayaVi2 with some options on the command line, thus loading d
```

Run MayaVi2 as is

Note: It is supposed here, in order to work, that your PATH variable contains the directory where !MayaVi2 binary is installed (for instance, /usr/local/bin).

Begin to run !MayaVi2:

```
mayavi2
```

!MayaVi2 will be launched with a TVTK Scene which is blank.

```
* Let's start by loading some data. Go into File/Open/Vtk file. This
```

```
The TVTK Scene has not changed, but you see that the file is loaded
```

```
* Now that we have data, we can visualize it. First lets create an o
```

```
Go into the menu item Visualize/Modules/Outline. This creates an Out
```

```
* Now lets add three grid planes.
```

```
Select Visualize/Modules/!GridPlane, this create a grid plane in the
```

```
* We can project the object onto the Grid planes using contour grid
```

```
* Now we can look at our data by selecting Visualize/Modules/Isosurf
```

```
* We can now finish adding a scalar cut plane by selecting Visualize
```

The result should look like this (for issue concerning background/forground colors, see [:Cookbook/MayaVi/Tips: Cookbook/MayaVi/Tips]):

[](files/attachments/MayaVi_RunningMayavi2/mv2.png

Run MayaVi2 with some parameters: modules and filters available

In order to know what you can set as parameters on the command line, type

```
mayavi2 -h
```

This will list all the available options. Amongst all these options are:

`* -d file.vtk: set the VTK file to load into !MayaVi2;`

`* -m ` *`module`* ` : set the module to use. The following list displays a`

`* ` *`Axes`* ` : draws a simple axes using tvtk.!CubeAxesActor;`

`* ` *`!ContourGridPlane`* ` : a contour grid plane module. This module let`

`* ` *`!CustomGridPlane`* ` : a custom grid plane with a lot more flexibili`

`* ` *`Glyph`* ` : displays different types of glyphs oriented and colored`

`* ` *`!GridPlane`* ` : a simple grid plane module;`

`* ` *`!ImagePlaneWidget`* ` : a simple !ImagePlaneWidget module to view im`

`* ` *`!IsoSurface`* ` : an !IsoSurface module that allows the user to make`

`* ` *`!OrientationAxes`* ` : creates a small axes on the side that indicat`

`* ` *`Outline`* ` : an outline module that draws an outline for the given`

`* ` *`!ScalarCutPlane`* ` : takes a cut plane of any input data set using`

`* ` *`!SliceUnstructuredGrid`* ` : this module takes a slice of the unstru`

`* ` *`Streamline`* ` : allows the user to draw streamlines for given vecto`

`* ` *`!StructuredGridOutline`* ` : draws a grid-conforming outline for str`

`* ` *`Surface`* ` : draws a surface for any input dataset with optional co`

`* ` *`Text`* ` : this module allows the user to place text on the screen;`

`* ` *`!VectorCutPlane`* ` : takes an arbitrary slice of the input data usi`

`* ` *`Vectors`* ` : displays different types of glyphs oriented and colore`

`* ` *`Volume`* ` : the Volume module visualizes scalar fields using volume`

`* ` *`!WarpVectorCutPlane`* ` : takes an arbitrary slice of the input data`

`* -f ` *`filter`* ` : set the filter to use (load it before module if you`

```
* !CellToPointData : transforms cell attribute data to point data
```

```
* Delaunay2D : performs a 2D Delaunay triangulation using the tvtk
```

```
* Delaunay3D : performs a 3D Delaunay triangulation using the tvtk
```

```
* !ExtractGrid : allows a user to select a part of a structured gr
```

```
* !ExtractUnstructuredGrid : allows a user to select a part of an
```

```
* !ExtractVectorNorm : computes the norm (Euclidean) of the input
```

```
* !MaskPoints : selectively passes the input points downstream. Th
```

```
* !PointToCellData : does the inverse of the !CellToPointData filt
```

```
* !PolyDataNormals : computes normals from input data. This gives
```

```
* Threshold : a simple filter that thresholds on input data;
```

```
* !TransformData : performs a linear transformation to input data
```

```
* !WarpScalar : warps the input data along a particular direction
```

```
* !WarpVector : warps the input data along a the point vector attr
```

Ok, you think you'll get rapidly tired to type all these long name modules and filters ? Don't worry, use your shell completion command !

For instance, for (t)csh shell, you can put this line in your configuration shell file:

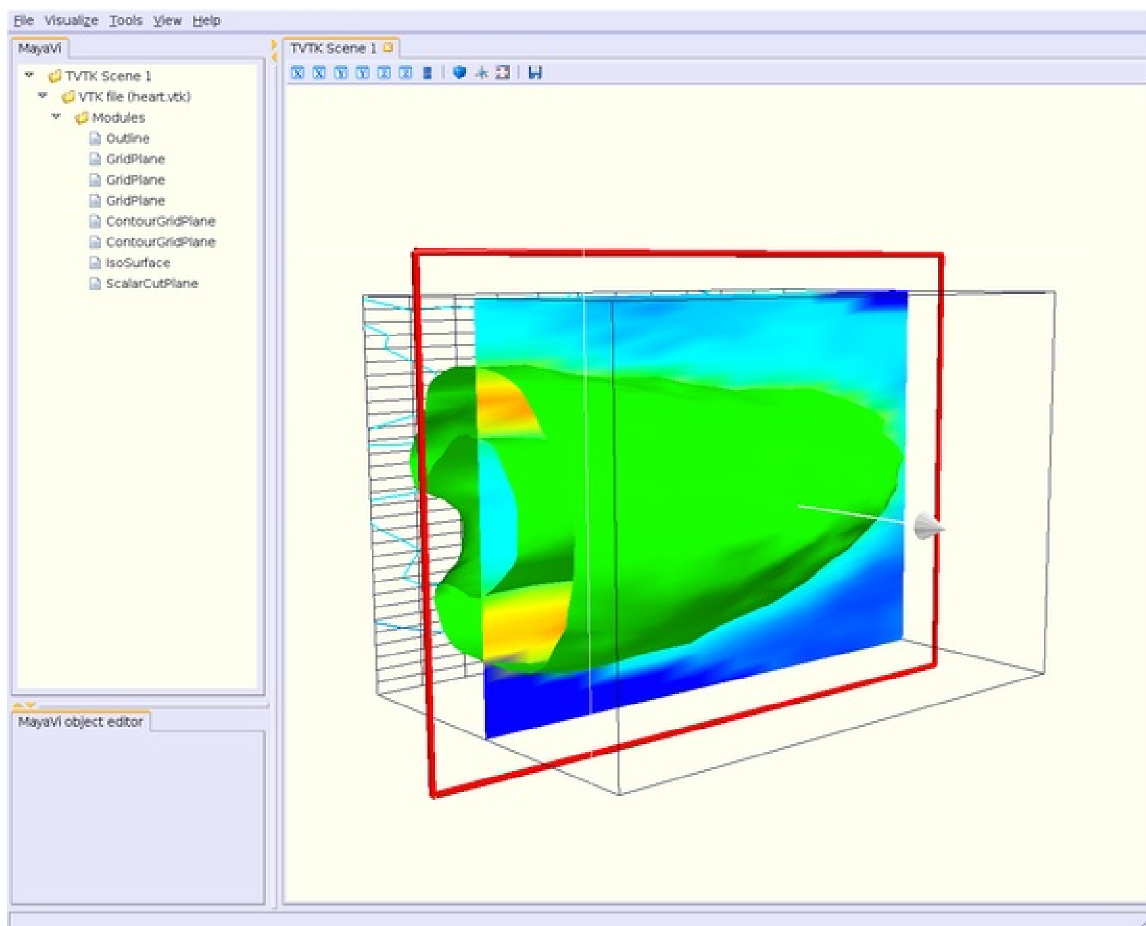
```
complete mayavi2 c/-/"(3 d f m n M p q w x z)"/ n/-3/f:*.3ds/ n/-d/
```

Quite long to type the first time, but once typed, no more effort load the module or filter you want to use ! ;-)

Thus, typing (in !MayaVi2's examples directory, see [:Cookbook/MayaVi/Installation: Cookbook/MayaVi/Installation]):

```
mayavi2 -d data/heart.vtk -m Outline -m GridPlane -m GridPlane -m C
```

you should get this:



Ok, this is not exactly the same as on the previous figure although exactly the same modules have been loaded.

The reason is that you did not set (and you can not, AFAIK) some modules properties, such as iso-value for !Iso Surface module, normals to the !GridPlane, etc.

Hopefully, you can set these parameters “by hand” using the GUI.

So now, let’s play with this GUI ;-)

Moving around

So, you can see on these two figures the render window on the right (TVTK Scene) beside the modules tree on the left. Let’s consider the first figure.

You can handle rendering scene with the mouse as usual using OpenGL:

- moving the mouse with the left button pressed rotates the scene;
- moving the mouse with middle button pressed translates it;
- and moving the mouse with right button pressed zooms in/out (note: you can also use the wheel of your mouse, if you get one).

Note: You can get a “predefined” angle of view (normal to x axis, y or z) clicking on the “View” item or on each small icons (first X: Ox axis points backwards you, second X: Ox axis points towards you, etc...)

On the left of the window, you can see which modules are loaded (“TVTK Scene” is the “root” of the scene, and “VTK file” is the data source, i.e. the heart.vtk file):

- “Outline” module displays the box surrounding the scene;
- you have three grid planes (”!GridPlane” module), at coordinates x = 0, y = 0, and z = 0;
- two contour grid planes (”!ContourGridPlane” module): the first displays contour lines (vertically), the second, the cutplane at z = const;
- ”!IsoSurface” module displays the heart as yellow;
- the last cutplane (vertically, at y = const) is displayed by the ”!ScalarCutPlane” module.

Note that no filter is used in these scenes.

Using each module/filter is very intuitive. Click on your module/filter in the tree and set some parameters with the mouse or enter some values at the keyboard, in the window at the bottom left of the GUI.

If you want to copy/paste/delete/cut a given module/filter, click on it with the right button. A small window popups, with a items list.

Note: You can get a larger window double-clicking on the choosen module/filter in the tree.

To load other modules or add filters, click on the “Visualize” item at the top of the window.

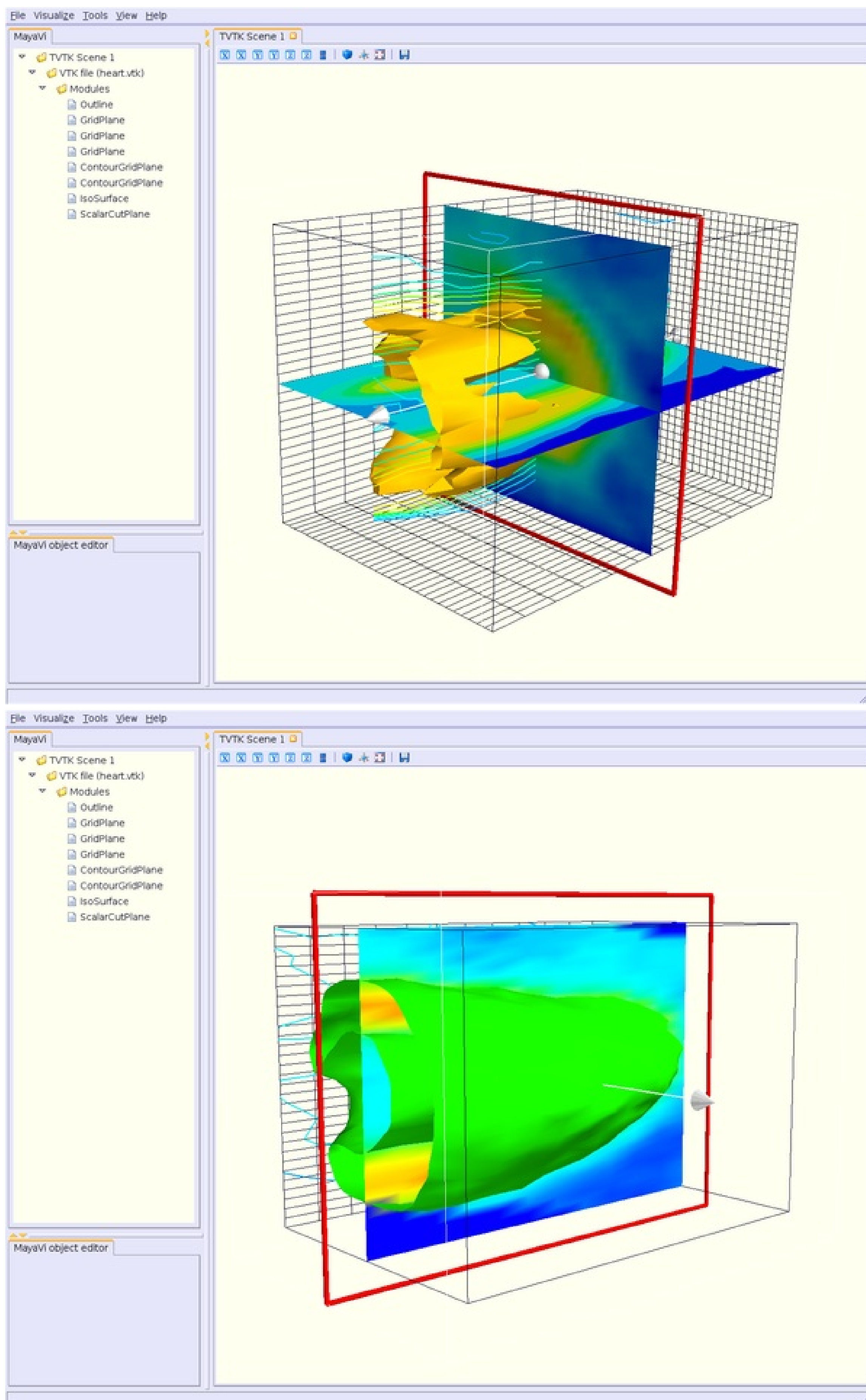
At last, you can also load another VTK data file, load/save the scene in a ”!MayaVi2” file (with extension .mv2), or convert it to the image format you want (PNG, JPEG...), clicking on the “File” item or on the appropriate icon (small floppy disk).

You can also get your scene in full-screen clicking on the small icon representing four red arrows in a little square. To disable full-screen mode, type ‘e’ or ‘q’.

This is the simplest way to use !MayaVi2. You are recalled here that you can also try ‘’mayavi2 -h’’ to see what options and arguments you can add to the !MayaVi2 command line.

Attachments

- `mv2.png`
- `mv2_cmdline.png`


File Visualize Tools View Help
MayaVi
TVTK Scene 1
VTK file (heart.vtk)
Modules
Outline
GridPlane
GridPlane
GridPlane
ContourGridPlane
ContourGridPlane
IsoSurface
ScalarCutPlane
MayaVi object editor
TVTK Scene 1
File Visualize Tools View Help
MayaVi
TVTK Scene 1
VTK file (heart.vtk)
Modules
Outline
GridPlane
GridPlane
GridPlane
ContourGridPlane
ContourGridPlane
IsoSurface
ScalarCutPlane
MayaVi object editor
TVTK Scene 1

Scripting Mayavi 2

||<#80FF80> This page presents scripting Mayavi2 using the advanced, object-oriented API. Mayavi2 has recently acquired an easy-to-use, thought maybe not as powerful, scripting module: mlab. You are invited to refer to the section of Mayavi2 user guide. Reading this page will give you a deeper understanding of how Mayavi2 works, and it complements the user guide. ||

Introduction

To script !MayaVi2, you need (at least):

```
* your favorite text editor;``* python installed ;-)``* !MayaVi2 ins
```

Scripting !MayaVi2 is quite simple because !MayaVi2 is written in python and based on TVTK, which eases the uses of all VTK objects.

In the following, you'll be learned how to script and use !MayaVi2 modules and filters.

Modules can be split in two parts:

```
* modules which do not interact with VTK data, and are seldom modifi
```

```
* modules which do interact with VTK data, and those you want to pla
```

Before starting, let's see the "main template" of a !MayaVi2 script written in python.

Main template: create your MayaVi2 class

A !MayaVi2 script should contain at least the following few lines:

```
#! /usr/bin/env python

from enthought.mayavi.app import Mayavi

class MyClass(Mayavi):

    def run(self):
        script = self.script
        # `self.script` is the MayaVi Script interface (an instance
        # enthought.mayavi.script.Script) that is created by the
        # base `Mayavi` class.  Here we save a local reference for
        # convenience.

        ## import any Mayavi modules and filters you want (they mus
        .../...

        script.new_scene()                      # to create the ren

        ## your stuff (modules, filters, etc) here
        .../...

if __name__ == '__main__':

    mc = MyClass()
    mc.main()
```

Adding modules or filters is quite simple: you have to import it, and then add it to your !MayaVi2 class.

To add a module, type:

```
from enthought.mayavi.modules.foo_module import FooModule
.../...
mymodule = FooModule()
script.add_module(mymodule)
```

To add a filter, type:

```
from enthought.mayavi.filters.bar_filter import BarFilter
.../...
myfilter = BarFilter()
script.add_filter(myfilter)
```

Notice the used syntax: for modules for example, foo_module is the foo_module python file (without the extension .py) in the subdirectory module/ of mayavi/ directory (lower case, underscore); this file contains the class FooModule (no underscore, capitalized name).

But first of all, before rendering your scene with the modules and the filters you want to use, you have to load some data, of course.

Loading data

You have the choice between:

```
* create a 3D data array, for scalars data (for vectors data, you ha
```

```
* load a data file with !FileReader methods.
```

Loading data from array using ArraySource method

For example, we will create a 50*50*50 3D (scalar) array of a product of cosinus & sinus functions.

To do this, we need to load the appropriate modules:

```
import scipy
from scipy import ogrid, sin, cos, sqrt, pi

from enthought.mayavi.sources.array_source import ArraySource

Nx = 50
Ny = 50
Nz = 50

Lx = 1
Ly = 1
Lz = 1

x, y, z = ogrid[0:Lx:(Nx+1)*1j,0:Ly:(Ny+1)*1j,0:Lz:(Nz+1)*1j]

# Strictly speaking, H is the magnetic field of the "transverse ele
# of a rectangular resonator cavity, with dimensions Lx, Ly, Lz
Hx = sin(x*pi/Lx)*cos(y*pi/Ly)*cos(z*pi/Lz)
Hy = cos(x*pi/Lx)*sin(y*pi/Ly)*cos(z*pi/Lz)
Hz = cos(x*pi/Lx)*cos(y*pi/Ly)*sin(z*pi/Lz)
Hv_scal = sqrt(Hx**2 + Hy**2 + Hz**2)

# We want to load a scalars data (Hv_scal) as magnitude of a given
# Hv_scal is a 3D scalars data, Hv is a 4D scalars data
src = ArraySource()
src.scalar_data = Hv_scal # load scalars data

# To load vectors data
# src.vector_data = Hv
```

Loading data from file using FileReader methods

To load a VTK data file, say heart.vtk file in mayavi/examples/data/ directory, simply type:

```
from enthought.mayavi.sources.vtk_file_reader import VTKFileReader

src = VTKFileReader()
src.initialize("heart.vtk")
```

Note: Files with .vtk extension are called “legacy VTK” files. !MayaVi2 can read a lot of other files formats (XML, files from Ensight, Plot3D and so on). For example, you can load an XML file (with extension .vti, .vtp, .vtr, .vts, .vtu, etc) using VTKXML!FileReader method.

Add the source to your MayaVi2 class

Then, once your data are loaded using one of the two methods above, add the source with the add_source() method in the body of the class !MyClass (after script.new_scene):

```
script.add_source(src)
```

The four basic modules Outline, Axes, !OrientationAxes and Text will be presented now.

Basic Modules

See the [:Cookbook/MayaVi/ScriptingMayavi2/BasicModules: Basic Modules] wiki page.

Main Modules

See the [:Cookbook/MayaVi/ScriptingMayavi2/MainModules: Main Modules] wiki page.

Filters

See the [:Cookbook/MayaVi/ScriptingMayavi2/Filters: Filters] wiki page.

Mayavi / TVTK

Mayabi: mlab

||<#80FF80> This page is about the tvtk.tools.mlab module. You are strongly advised to use the more recent mayavi.mlab module which can also be used from *ipython -wthread* or as a library inside another application.||

The `mlab.py` module allows for simple 3D plotting of objects. It provides an object oriented approch to 3d visualization.

It relies on the simple TVTK module, rather than on the full blown Mayavi application. As a result it has less dependencies. However, it is harder to extend and more limited. The developers are not planning any feature addition to this module, and we strongly advise you to use the Mayavi mlab module, which provides the same usecases, but with more functionalities and is under constant development. For more information, you can read the relevant section of the Mayavi user guide

‘’’ Table of Contents ‘’‘

A Simple Example

||<#FF8080> **Important:** All these examples must be run in “ipython -wthread” or in a Wx application (like pycrust, or the Mayavi2 application). They **will not work** if you don’t use this option.||

Start with `ipython -wthread` and paste the following code:

```
.. code:: python
```

> import scipy
>
> # prepare some interesting function: def f(x, y):
>
> > return 3.0*scipy.sin(*x*y+1e-4)/(x*y+1e-4)
>
> x = scipy.arange(-7., 7.05, 0.1) y = scipy.arange(-5., 5.05, 0.1)
>
> # 3D visualization of f: from enthought.tvtk.tools import mlab fig = mlab.figure() s = mlab.SurfRegular(x, y, f) fig.add(s)

[](files/attachments/MayaVi_mlab/simple_example.png

Changing axis and colour

```
from scipy import *

[x,y]=mgrid[-5:5:0.1,-5:5:0.1]
r=sqrt(x**2+y**2)
z=sin(3*r)/(r)

from enthought.tvtk.tools import mlab

# Open a viewer without the object browser:
f=mlab.figure(browser=False)

s=mlab.Surf(x,y,z,z)
f.add(s)
s.scalar_bar.title='sinc(r)'
s.show_scalar_bar=True
# LUT means "Look-Up Table", it give the mapping between scalar val
s.lut_type='blue-red'
# The current figure has two objects, the outline object originaly
# and the surf object that we added.
f.objects[0].axis.z_label='value'
t=mlab.Title()
t.text='Sampling function'
f.add(t)
# Edit the title properties with the GUI:
t.edit_traits()
```

[](files/attachments/MayaVi_mlab/tvtk.mlab_example.png)

List of different functionalities

The implementation provided here is object oriented and each visualization capability is implemented as a class that has traits. So each of these may be configured. Each visualization class derives (ultimately) from MLabBase which is responsible for adding/removing its actors into the render window. The classes all require that the RenderWindow be a `pyface.tvtk.scene.Scene` instance (this constraint can be relaxed if necessary later on).

This module offers the following broad class of functionality:

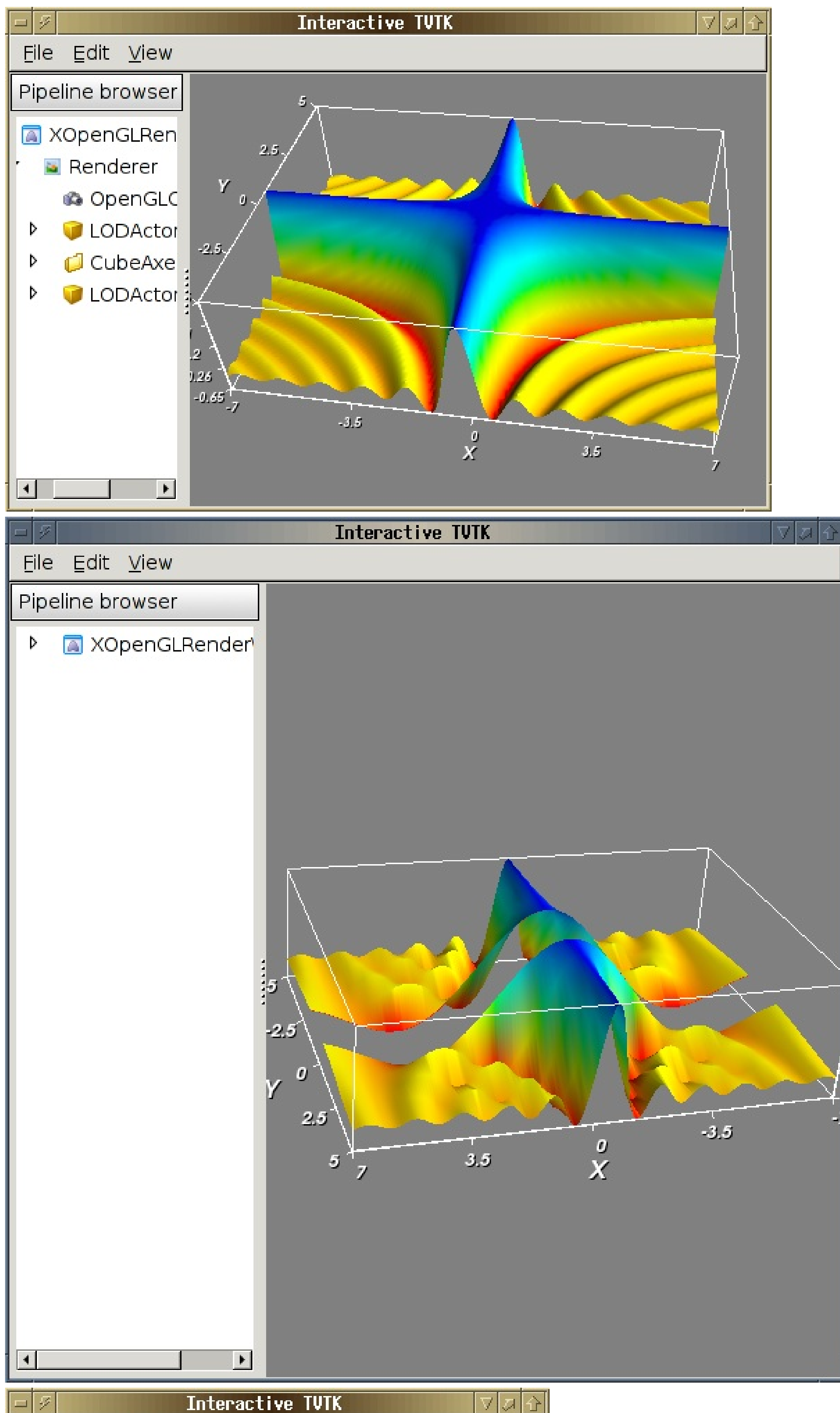
` * ` **`Figure`** ` This basically manages all of the objects rendered. Just` figure `may be used to create a nice Figure instance.``* ` **`Glyphs`** ` This` Arrows ` , ` Cones ` , ` Cubes ` , `` ` Cylinders ` , ` Spheres ` , and ` Points ` .``* ` **`Li`** mayavi.tools.imv.viewi ` . `

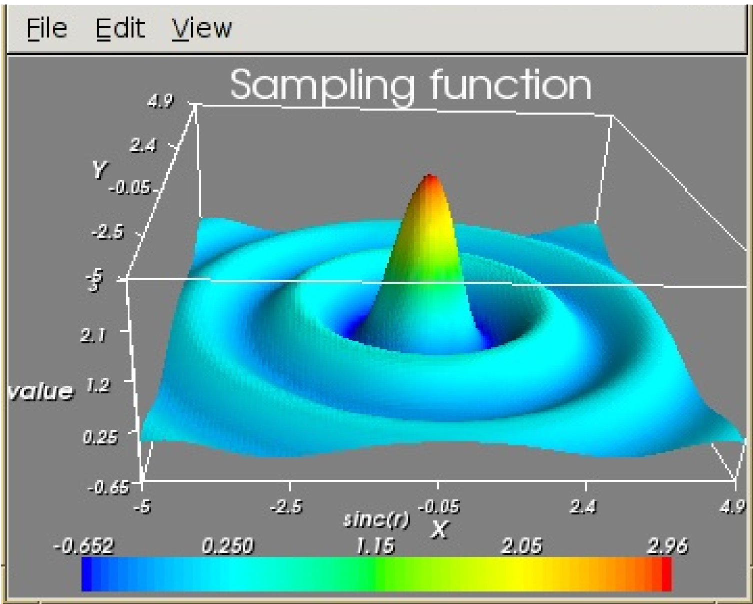
To see nice examples of all of these look at the `test_*` functions at the end of this file. Here is a quick example that uses these test functions:

```
from enthought.tvtk.tools import mlab
f = mlab.figure()
mlab.test_surf(f) # Create a spherical harmonic.
f.pop() # Remove it.
mlab.test_molecule(f) # Show a caffeine molecule.
f.renwin.reset_zoom() # Scale the view.
f.pop() # Remove this.
mlab.test_lines(f) # Show pretty lines.
f.clear() # Remove all the stuff on screen.
```

Attachments

- `simple_example.png`
- `simpleexample.png`
- `tvtk.mlab_example.png`


Interactive TVTK
File Edit View
Pipeline browser
XOpenGLRen
Renderer
OpenGLC
LODActor
CubeAxe
LODActor
Y
X
Interactive TVTK
File Edit View
Pipeline browser
XOpenGLRender
Y
X
Interactive TVTK


File Edit View
Sampling function
Y
value
X
sinc(r)
-0.652
0.250
1.15
2.05
2.96

Mayavi tvtk

||<#80FF80> This page is not the main source of documentation. You are invited to refer to the Mayavi2 home page for up-to-date documentation on TVTK. In particular, bear in mind that if you are looking for a high-level Python 3D plotting library, Mayavi also provides the right API, and can be embedded (see the user guide). ||

Numpy & Scipy / Advanced topics

Views versus copies in NumPy

From time to time, people write to the !NumPy list asking in which cases a view of an array is created and in which it isn't. This page tries to clarify some tricky points on this rather subtle subject.

What is a view of a NumPy array?

As its name is saying, it is simply another way of **viewing** the data of the array. Technically, that means that the data of both objects is *shared*. You can create views by selecting a **slice** of the original array, or also by changing the **dtype** (or a combination of both). These different kinds of views are described below.

Slice views

This is probably the most common source of view creations in !NumPy. The rule of thumb for creating a slice view is that the viewed elements can be addressed with offsets, strides, and counts in the original array. For example:

```
>>> a = numpy.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> v1 = a[1:2]
>>> v1
array([1])
>>> a[1] = 2
>>> v1
array([2])
>>> v2 = a[1::3]
>>> v2
array([2, 4, 7])
>>> a[7] = 10
>>> v2
array([ 2,  4, 10])
```

In the above code snippet, () and () are views of , because if you update , and will reflect the change.

Dtype views

Another way to create array views is by assigning another **dtype** to the same data area. For example:

```
>>> b = numpy.arange(10, dtype='int16')
>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int16)
>>> v3 = b.view('int32')
>>> v3 += 1
>>> b
array([1, 1, 3, 3, 5, 5, 7, 7, 9, 9], dtype=int16)
>>> v4 = b.view('int8')
>>> v4
array([1, 0, 1, 0, 3, 0, 3, 0, 5, 0, 5, 0, 7, 0, 7, 0, 9, 0, 9, 0],
```

In that case, and are views of . As you can see, **dtype views** are not as useful as **slice views**, but can be handy in some cases (for example, for quickly looking at the bytes of a generic array).

FAQ

I think I understand what a view is, but why fancy indexing is not returning a view?

One might be tempted to think that doing fancy indexing may lead to sliced views. But this is not true:

```
>>> a = numpy.arange(10)
>>> c1 = a[[1,3]]
>>> c2 = a[[3,1,1]]
>>> a[:] = 100
>>> c1
array([1, 3])
>>> c2
array([3, 1, 1])
```

The reason why a fancy indexing is not returning a view is that, in general, it cannot be expressed as a **slice** (in the sense stated above of being able to be addressed with offsets, strides, and counts).

For example, fancy indexing for could have been expressed by , but it is not possible to do the same for by means of a slice. So, this is why an object with a *copy* of the original data is returned instead.

But fancy indexing does seem to return views sometimes, doesn’t it?

Many users get fooled and think that fancy indexing returns views instead of copies when they use this idiom:

```
>>> a = numpy.arange(10)
>>> a[[1,2]] = 100
>>> a
array([  0, 100, 100,   3,   4,   5,   6,   7,   8,   9])
```

So, it seems that a<1,2> is actually a *view* because elements 1 and 2 have been updated. However, if we try this step by step, it won’t work:

```
>>> a = numpy.arange(10)
>>> c1 = a[[1,2]]
>>> c1[:] = 100
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> c1
array([100, 100])
```

What happens here is that, in the first idiom (), the python interpreter translate it to:

```
a.__setitem__([1,2], 100)
```

i.e. there is not need to create neither a view or a copy because the method can be evaluated *inplace* (i.e. no new object creation is involved).

However, the second idiom () is translated to:

```
c1 = a.__getitem__([1,2])
c1.__setitem__(slice(None, None, None), 100)  # [:] translates int
```

i.e. a new object with a **copy** (remember, fancy indexing does not return views) of some elements of is created and returned prior to call .

The rule of thumb here can be: in the context of **lvalue indexing** (i.e. the indices are placed in the left hand side value of an assignment), no view or copy of the array is created (because there is no need to). However, with regular values, the above rules for creating views does apply.

A final exercise

Finally, let's put a somewhat advanced problem. The next snippet works as expected:

```
>>> a = numpy.arange(12).reshape(3,4)
>>> ifancy = [0,2]
>>> islice = slice(0,3,2)
>>> a[islice, :][:, ifancy] = 100
>>> a
array([[100,   1, 100,   3],
 [  4,   5,   6,   7],
 [100,   9, 100,  11]])
```

but the next one does not:

```
>>> a = numpy.arange(12).reshape(3,4)
>>> ifancy = [0,2]
>>> islice = slice(0,3,2)
>>> a[ifancy, :][:, islice] = 100  # note that ifancy and islice ar
>>> a
array([[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11]])
```

Would the reader discover why is the difference? ''Hint: think in terms of the sequence of {{{**getitem**()}}} and {{{**setitem**()}}} calls and what they do on each example.''

Numpy & Scipy / Interpolation

Interpolation

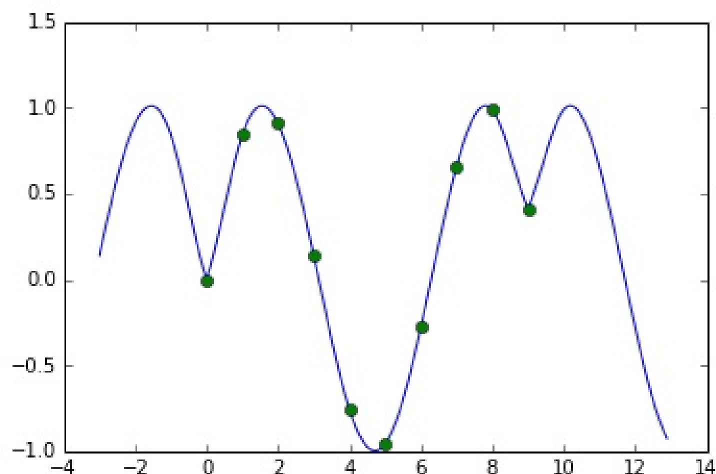
Using B-splines in scipy.signal

Example showing how to use B-splines in scipy.signal to do interpolation. The input points must be equally spaced to use these routine.

```
from numpy import r_, sin
from scipy.signal import cspline1d, cspline1d_eval
%pylab inline

x = r_[0:10]
dx = x[1]-x[0]
newx = r_[-3:13:0.1]  # notice outside the original domain
y = sin(x)
cj = cspline1d(y)
newy = cspline1d_eval(cj, newx, dx=dx,x0=x[0])
from pylab import plot, show
plot(newx, newy, x, y, 'o')
show()
```

```
Populating the interactive namespace from numpy and matplotlib
```



N-D interpolation for equally-spaced data

The scipy.ndimage package also contains spline_filter and map_coordinates which can be used to perform N-dimensional interpolation for equally-spaced data. A two-dimensional example is given below:

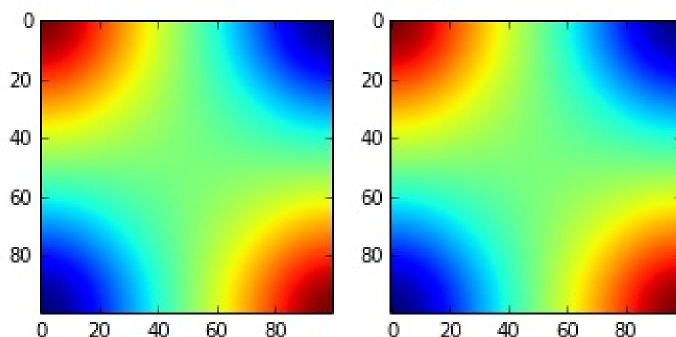
```
from scipy import ogrid, sin, mgrid, ndimage, array
from matplotlib import pyplot as plt

x,y = ogrid[-1:1:5j,-1:1:5j]
fvals = sin(x)*sin(y)
newx,newy = mgrid[-1:1:100j,-1:1:100j]
x0 = x[0,0]
y0 = y[0,0]
dx = x[1,0] - x0
dy = y[0,1] - y0
ivals = (newx - x0)/dx
jvals = (newy - y0)/dy
coords = array([ivals, jvals])
newf1 = ndimage.map_coordinates(fvals, coords)
```

To pre-compute the weights (for multiple interpolation results), you would use

```
coeffs = ndimage.spline_filter(fvals)
newf2 = ndimage.map_coordinates(coeffs, coords, prefilter=False)

plt.subplot(1,2,1)
plt.imshow(newf1)
plt.subplot(1,2,2)
plt.imshow(newf2)
plt.show()
```



Interpolation of an N-D curve

The scipy.interpolate packages wraps the netlib FITPACK routines (Dierckx) for calculating smoothing splines for various kinds of data and geometries. Although the data is evenly spaced in this example, it need not be so to use this routine.

```
from numpy import arange, cos, linspace, pi, sin, random
from scipy.interpolate import splprep, splev

# make ascending spiral in 3-space
t=linspace(0,1.75*2*pi,100)

x = sin(t)
y = cos(t)
z = t

# add noise
x+= random.normal(scale=0.1, size=x.shape)
y+= random.normal(scale=0.1, size=y.shape)
z+= random.normal(scale=0.1, size=z.shape)

# spline parameters
s=3.0 # smoothness parameter
k=2 # spline order
nest=-1 # estimate of number of knots needed (-1 = maximal)

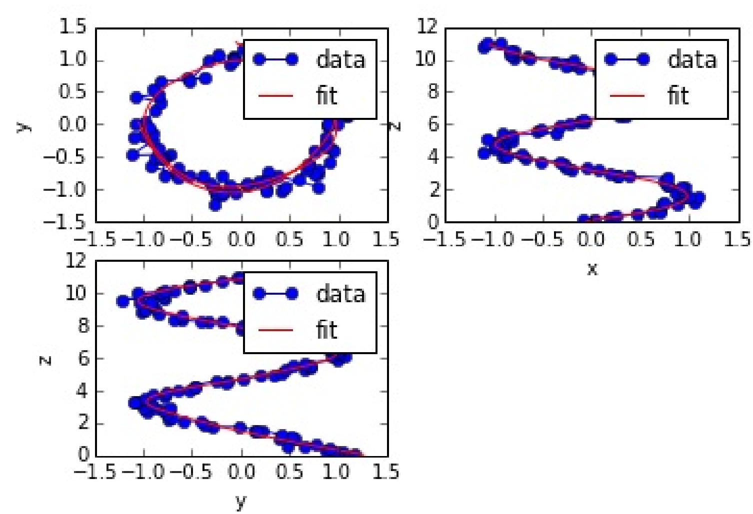
# find the knot points
tckp,u = splprep([x,y,z],s=s,k=k,nest=-1)

# evaluate spline, including interpolated points
xnew,ynew,znew = splev(linspace(0,1,400),tckp)

import pylab
pylab.subplot(2,2,1)
data,=pylab.plot(x,y,'bo-',label='data')
fit,=pylab.plot(xnew,ynew,'r-',label='fit')
pylab.legend()
pylab.xlabel('x')
pylab.ylabel('y')

pylab.subplot(2,2,2)
data,=pylab.plot(x,z,'bo-',label='data')
fit,=pylab.plot(xnew,znew,'r-',label='fit')
pylab.legend()
pylab.xlabel('x')
pylab.ylabel('z')

pylab.subplot(2,2,3)
data,=pylab.plot(y,z,'bo-',label='data')
fit,=pylab.plot(ynew,znew,'r-',label='fit')
pylab.legend()
pylab.xlabel('y')
pylab.ylabel('z')
plt.show()
```


data
fit
y
z
x
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
12
10
8
6
4
2
0
−1.5 −1.0 −0.5 0.0 0.5 1.0 1.5

Using radial basis functions for smoothing/interpolation

Radial basis functions can be used for smoothing/interpolating scattered data in n-dimensions, but should be used with caution for extrapolation outside of the observed data range.

1d example

This example compares the usage of the Rbf and UnivariateSpline classes from the scipy.interpolate module.

```
import numpy as np
from scipy.interpolate import Rbf, InterpolatedUnivariateSpline

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

# setup data
x = np.linspace(0, 10, 9)
y = np.sin(x)
xi = np.linspace(0, 10, 101)

# use fitpack2 method
ius = InterpolatedUnivariateSpline(x, y)
yi = ius(xi)

plt.subplot(2, 1, 1)
plt.plot(x, y, 'bo')
plt.plot(xi, yi, 'g')
plt.plot(xi, np.sin(xi), 'r')
plt.title('Interpolation using univariate spline')

# use RBF method
rbf = Rbf(x, y)
fi = rbf(xi)

plt.subplot(2, 1, 2)
plt.plot(x, y, 'bo')
plt.plot(xi, yi, 'g')
plt.plot(xi, np.sin(xi), 'r')
plt.title('Interpolation using RBF - multiquadrics')
plt.savefig('rbf1d.png')
```

[](files/attachments/RadialBasisFunctions/rbf1dnew.png

Numpy & Scipy / Linear Algebra

Rank and nullspace of a matrix

The following module, rank_nullspace.py, provides the functions rank() and nullspace(). (Note that !NumPy already provides the function matrix_rank(); the function given here allows an absolute tolerance to be specified along with a relative tolerance.)

rank_nullspace.py

```
#!python
import numpy as np
from numpy.linalg import svd

def rank(A, atol=1e-13, rtol=0):
    """Estimate the rank (i.e. the dimension of the nullspace) of a

 The algorithm used by this function is based on the singular value
 decomposition of `A`.

 Parameters
 ----------
 A : ndarray
 A should be at most 2-D.  A 1-D array with length n will be treate
 as a 2-D with shape (1, n)
 atol : float
 The absolute tolerance for a zero singular value.  Singular values
 smaller than `atol` are considered to be zero.
 rtol : float
 The relative tolerance.  Singular values less than rtol*smax are
 considered to be zero, where smax is the largest singular value.

 If both `atol` and `rtol` are positive, the combined tolerance is
 maximum of the two; that is::
 tol = max(atol, rtol * smax)
 Singular values smaller than `tol` are considered to be zero.

 Return value
 ------------
 r : int
 The estimated rank of the matrix.

 See also
 --------
 numpy.linalg.matrix_rank
 matrix_rank is basically the same as this function, but it does no
 provide the option of the absolute tolerance.
 """

    A = np.atleast_2d(A)
```

```
    s = svd(A, compute_uv=False)
    tol = max(atol, rtol * s[0])
    rank = int((s >= tol).sum())
    return rank

def nullspace(A, atol=1e-13, rtol=0):
    """Compute an approximate basis for the nullspace of A.

 The algorithm used by this function is based on the singular value
 decomposition of `A`.

 Parameters
 ----------
 A : ndarray
 A should be at most 2-D.  A 1-D array with length k will be treate
 as a 2-D with shape (1, k)
 atol : float
 The absolute tolerance for a zero singular value.  Singular values
 smaller than `atol` are considered to be zero.
 rtol : float
 The relative tolerance.  Singular values less than rtol*smax are
 considered to be zero, where smax is the largest singular value.

 If both `atol` and `rtol` are positive, the combined tolerance is
 maximum of the two; that is::
 tol = max(atol, rtol * smax)
 Singular values smaller than `tol` are considered to be zero.

 Return value
 ------------
 ns : ndarray
 If `A` is an array with shape (m, k), then `ns` will be an array
 with shape (k, n), where n is the estimated dimension of the
 nullspace of `A`.  The columns of `ns` are a basis for the
 nullspace; each element in numpy.dot(A, ns) will be approximately
 zero.
 """

    A = np.atleast_2d(A)
    u, s, vh = svd(A)
    tol = max(atol, rtol * s[0])
    nnz = (s >= tol).sum()
    ns = vh[nnz:].conj().T
    return ns
```

Here’s a demonstration script.

```
#!python
import numpy as np

from rank_nullspace import rank, nullspace

def checkit(a):
    print "a:"
    print a
    r = rank(a)
    print "rank is", r
    ns = nullspace(a)
    print "nullspace:"
    print ns
    if ns.size > 0:
        res = np.abs(np.dot(a, ns)).max()
        print "max residual is", res

print "-"*25

a = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
checkit(a)

print "-"*25

a = np.array([[0.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
checkit(a)

print "-"*25

a = np.array([[0.0, 1.0, 2.0, 4.0], [1.0, 2.0, 3.0, 4.0]])
checkit(a)

print "-"*25

a = np.array([[1.0,   1.0j,   2.0+2.0j],
              [1.0j, -1.0,   -2.0+2.0j],
              [0.5,   0.5j,   1.0+1.0j]])
checkit(a)

print "-"*25
```

And here is the output of the script.

```
-------------------------
a:
[[ 1\.  2\.  3.]
 [ 4\.  5\.  6.]
 [ 7\.  8\.  9.]]
rank is 2
nullspace:
[[-0.40824829]
 [ 0.81649658]
 [-0.40824829]]
max residual is 4.4408920985e-16
-------------------------
a:
[[ 0\.  2\.  3.]
 [ 4\.  5\.  6.]
 [ 7\.  8\.  9.]]
rank is 3
nullspace:
[]
-------------------------
a:
[[ 0\.  1\.  2\.  4.]
 [ 1\.  2\.  3\.  4.]]
rank is 2
nullspace:
[[-0.48666474 -0.61177492]
 [-0.27946883  0.76717915]
 [ 0.76613356 -0.15540423]
 [-0.31319957 -0.11409267]]
max residual is 3.88578058619e-16
-------------------------
a:
[[ 1.0+0.j   0.0+1.j   2.0+2.j ]
 [ 0.0+1.j  -1.0+0.j  -2.0+2.j ]
 [ 0.5+0.j   0.0+0.5j  1.0+1.j ]]
rank is 1
nullspace:
[[ 0.00000000-0.j          -0.94868330-0.j         ]
 [ 0.13333333+0.93333333j  0.00000000-0.10540926j]
 [ 0.20000000-0.26666667j  0.21081851-0.21081851j]]
max residual is 1.04295984227e-15
-------------------------
```

Numpy & Scipy / Matplotlib

Histograms

Here is an example for creating a 2D histogram with variable bin size and displaying it.

```
#!/usr/bin/python
import numpy as np
import matplotlib as ml
import matplotlib.pyplot as plt

# First we define the bin edges
xedges = [0, 1, 1.5, 3, 5]
yedges = [0, 2, 3, 4, 6]

# Next we create a histogram H with random bin content
x = np.random.normal(3, 1, 100)
y = np.random.normal(1, 1, 100)
H, xedges, yedges = np.histogram2d(y, x, bins=(xedges, yedges))

# Or we fill the histogram H with a determined bin content
H = np.ones((4, 4)).cumsum().reshape(4, 4)
print H[::-1]  # This shows the bin content in the order as plottec

ml.rcParams['image.cmap'] = 'gist_heat'

fig = plt.figure(figsize=(6, 3.2))

# pcolormesh is useful for displaying exact bin edges
ax = fig.add_subplot(131)
ax.set_title('pcolormesh:\nexact bin edges')
X, Y = np.meshgrid(xedges, yedges)
plt.pcolormesh(X, Y, H)
ax.set_aspect('equal')

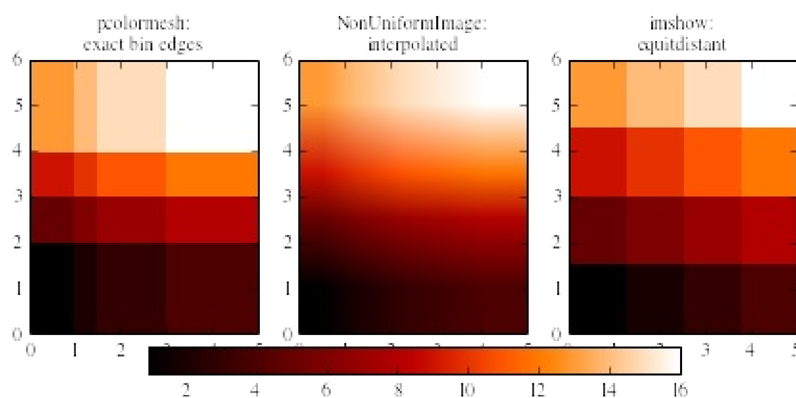
# NonUniformImage can be used for interpolation
ax = fig.add_subplot(132)
ax.set_title('NonUniformImage:\ninterpolated')
im = ml.image.NonUniformImage(ax, interpolation='bilinear')
xcenters = xedges[:-1] + 0.5 * (xedges[1:] - xedges[:-1])
ycenters = yedges[:-1] + 0.5 * (yedges[1:] - yedges[:-1])
im.set_data(xcenters, ycenters, H)
ax.images.append(im)
ax.set_xlim(xedges[0], xedges[-1])
ax.set_ylim(yedges[0], yedges[-1])
ax.set_aspect('equal')

# Imshow is useful for a simple equidistant representation of bin c
ax = fig.add_subplot(133)
ax.set_title('imshow:\nequitdistant')
```

```
im = plt.imshow(H, interpolation='nearest', origin='low',
                extent=[xedges[0], xedges[-1], yedges[0], yedges[-1

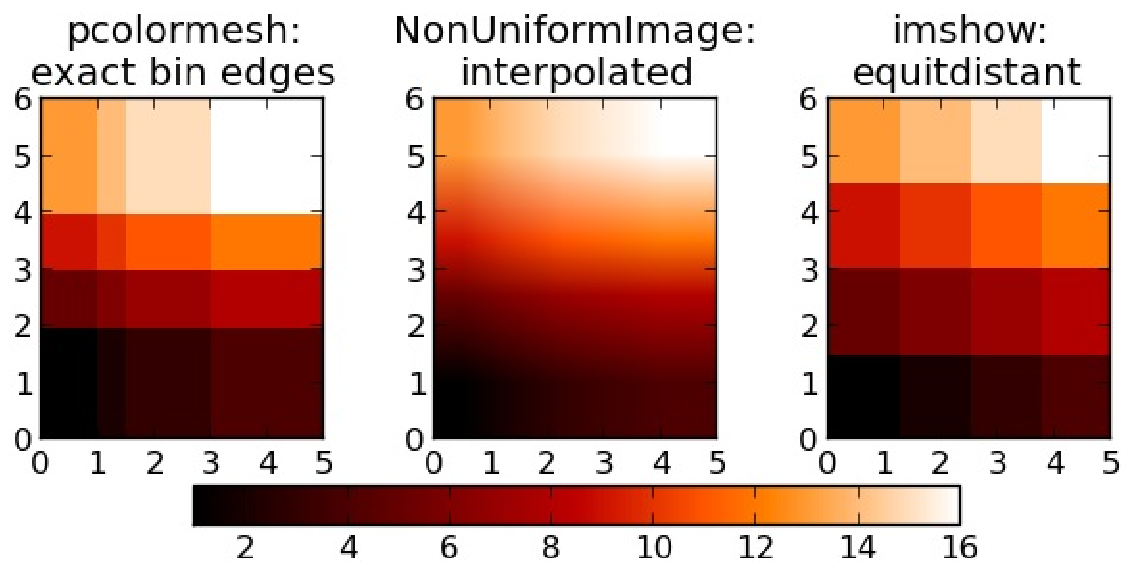
# Finally we add a color bar
cax = fig.add_axes([0.12, 0.1, 0.78, 0.4])
cax.get_xaxis().set_visible(False)
cax.get_yaxis().set_visible(False)
cax.patch.set_alpha(0)
cax.set_frame_on(False)
plt.colorbar(orientation='horizontal', ax=cax)
plt.tight_layout()
plt.show()
```

```
[[ 13\.  14\.  15\.  16.]
 [  9\.  10\.  11\.  12.]
 [  5\.   6\.   7\.   8.]
 [  1\.   2\.   3\.   4.]]
```



Attachments

- `histogram2d.png`


pcolormesh:
exact bin edges
NonUniformImage:
interpolated
imshow:
equitdistant
0 1 2 3 4 5
0 1 2 3 4 5 6
2 4 6 8 10 12 14 16

Numpy & Scipy / Optimization and fitting techniques

Fitting data

This page shows you how to fit experimental data and plots the results using matplotlib.

Fit examples with sinusoidal functions

Generating the data

Using real data is much more fun, but, just so that you can reproduce this example I will generate data to fit

```
import numpy as np
from numpy import pi, r_
import matplotlib.pyplot as plt
from scipy import optimize

# Generate data points with noise
num_points = 150
Tx = np.linspace(5., 8., num_points)
Ty = Tx

tX = 11.86*np.cos(2*pi/0.81*Tx-1.32) + 0.64*Tx+4*((0.5-np.random.ra
tY = -32.14*np.cos(2*np.pi/0.8*Ty-1.94) + 0.15*Ty+7*((0.5-np.randor
```

Fitting the data

We now have two sets of data: Tx and Ty, the time series, and tX and tY, sinusoidal data with noise. We are interested in finding the frequency of the sine wave.

```
# Fit the first set
fitfunc = lambda p, x: p[0]*np.cos(2*np.pi/p[1]*x+p[2]) + p[3]*x #
errfunc = lambda p, x, y: fitfunc(p, x) - y # Distance to the targe
p0 = [-15., 0.8, 0., -1.] # Initial guess for the parameters
p1, success = optimize.leastsq(errfunc, p0[:], args=(Tx, tX))

time = np.linspace(Tx.min(), Tx.max(), 100)
plt.plot(Tx, tX, "ro", time, fitfunc(p1, time), "r-") # Plot of the

# Fit the second set
p0 = [-15., 0.8, 0., -1.]
p2,success = optimize.leastsq(errfunc, p0[:], args=(Ty, tY))

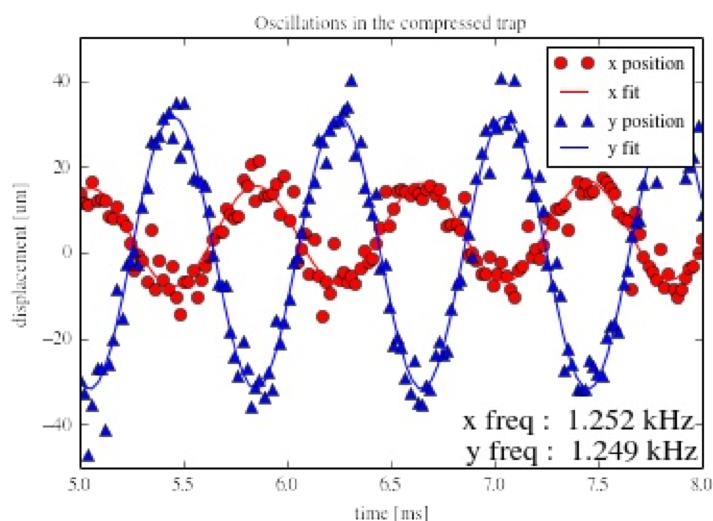
time = np.linspace(Ty.min(), Ty.max(), 100)
plt.plot(Ty, tY, "b^", time, fitfunc(p2, time), "b-")

# Legend the plot
plt.title("Oscillations in the compressed trap")
plt.xlabel("time [ms]")
plt.ylabel("displacement [um]")
plt.legend(('x position', 'x fit', 'y position', 'y fit'))

ax = plt.axes()

plt.text(0.8, 0.07,
         'x freq : %.3f kHz \n y freq : %.3f kHz' % (1/p1[1],1/p2[1
         fontsize=16,
         horizontalalignment='center',
         verticalalignment='center',
         transform=ax.transAxes)

plt.show()
```



A clever use of the cost function

Suppose that you have the same data set: two time-series of oscillating phenomena, but that you know that the frequency of the two oscillations is the same. A clever use of the cost function can allow you to fit both set of data in one fit, using the same frequency. The idea is that you return, as a “cost” array, the concatenation of the costs of your two data sets for one choice of parameters. Thus the leastsq routine is optimizing both data sets at the same time.

```
# Target function
fitfunc = lambda T, p, x: p[0]*np.cos(2*np.pi/T*x+p[1]) + p[2]*x
# Initial guess for the first set's parameters
p1 = r_[-15., 0., -1.]
# Initial guess for the second set's parameters
p2 = r_[-15., 0., -1.]
# Initial guess for the common period
T = 0.8
# Vector of the parameters to fit, it contains all the parameters o
p = r_[T, p1, p2]
# Cost function of the fit, compare it to the previous example.
errfunc = lambda p, x1, y1, x2, y2: r_[
                fitfunc(p[0], p[1:4], x1) - y1,
                fitfunc(p[0], p[4:7], x2) - y2
            ]
# This time we need to pass the two sets of data, there are thus fo
p,success = optimize.leastsq(errfunc, p, args=(Tx, tX, Ty, tY))
time = np.linspace(Tx.min(), Tx.max(), 100) # Plot of the first dat
plt.plot(Tx, tX, "ro", time, fitfunc(p[0], p[1:4], time),"r-")

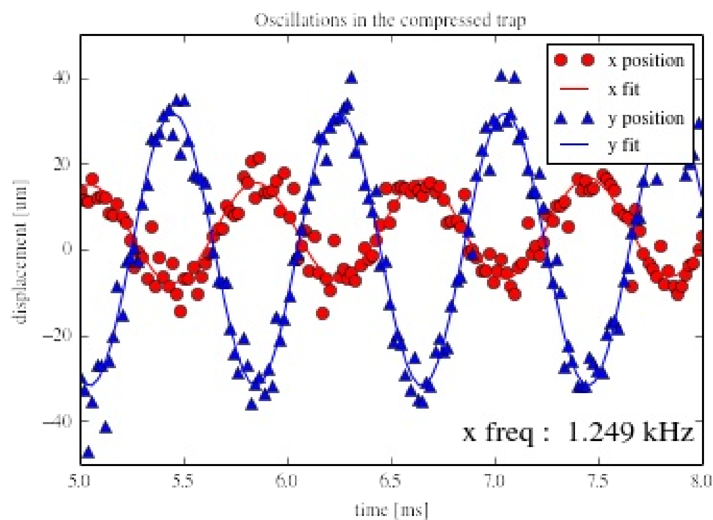
# Plot of the second data and the fit
time = np.linspace(Ty.min(), Ty.max(),100)
plt.plot(Ty, tY, "b^", time, fitfunc(p[0], p[4:7], time),"b-")

# Legend the plot
plt.title("Oscillations in the compressed trap")
plt.xlabel("time [ms]")
plt.ylabel("displacement [um]")
plt.legend(('x position', 'x fit', 'y position', 'y fit'))

ax = plt.axes()

plt.text(0.8, 0.07,
         'x freq : %.3f kHz' % (1/p[0]),
         fontsize=16,
         horizontalalignment='center',
         verticalalignment='center',
         transform=ax.transAxes)
```

```
<matplotlib.text.Text at 0x7fab996e5b90>
```



Simplifying the syntax

Especially when using fits for interactive use, the standard syntax for optimize.leastsq can get really long. Using the following script can simplify your life:

```
import numpy as np
from scipy import optimize

class Parameter:
    def __init__(self, value):
            self.value = value

    def set(self, value):
            self.value = value

    def __call__(self):
            return self.value

def fit(function, parameters, y, x = None):
    def f(params):
        i = 0
        for p in parameters:
            p.set(params[i])
            i += 1
        return y - function(x)

    if x is None: x = np.arange(y.shape[0])
    p = [param() for param in parameters]
    return optimize.leastsq(f, p)
```

Now fitting becomes really easy, for example fitting to a gaussian:

```
# giving initial parameters
mu = Parameter(7)
sigma = Parameter(3)
height = Parameter(5)

# define your function:
def f(x): return height() * np.exp(-((x-mu())/sigma())**2)

# fit! (given that data is an array with the data to fit)
data = 10*np.exp(-np.linspace(0, 10, 100)**2) + np.random.rand(100)
print fit(f, [mu, sigma, height], data)
```

```
(array([ -1.7202343 ,  12.29906459,  10.74194291]), 1)
```

Fitting gaussian-shaped data

Calculating the moments of the distribution

Fitting gaussian-shaped data does not require an optimization routine. Just calculating the moments of the distribution is enough, and this is much faster.

However this works only if the gaussian is not cut out too much, and if it is not too small.

```
gaussian = lambda x: 3*np.exp(-(30-x)**2/20.)

data = gaussian(np.arange(100))

plt.plot(data, '.')

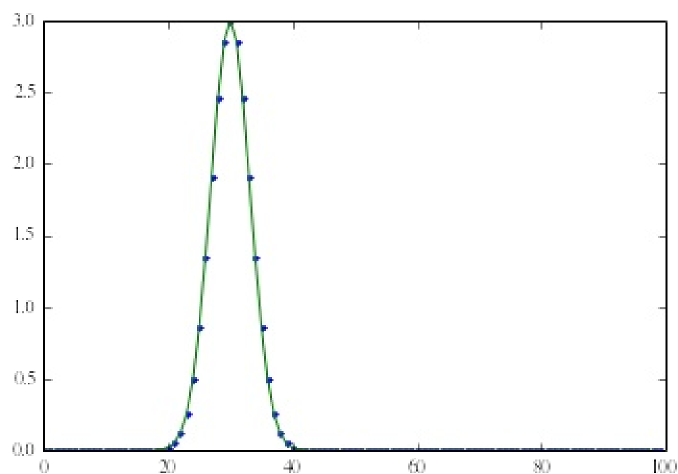
X = np.arange(data.size)
x = np.sum(X*data)/np.sum(data)
width = np.sqrt(np.abs(np.sum((X-x)**2*data)/np.sum(data)))

max = data.max()

fit = lambda t : max*np.exp(-(t-x)**2/(2*width**2))

plt.plot(fit(X), '-')
```

```
[<matplotlib.lines.Line2D at 0x7fab9977d990>]
```



Fitting a 2D gaussian

Here is robust code to fit a 2D gaussian. It calculates the moments of the data to guess the initial parameters for an optimization routine. For a more complete gaussian, one with an optional additive constant and rotation, see http://code.google.com/p/agpy/source/browse/trunk/agpy/gaussfitter.py. It also allows the specification of a known error.

```
def gaussian(height, center_x, center_y, width_x, width_y):
    """Returns a gaussian function with the given parameters"""
    width_x = float(width_x)
    width_y = float(width_y)
    return lambda x,y: height*np.exp(
                -(((center_x-x)/width_x)**2+((center_y-y)/width_y)*

def moments(data):
    """Returns (height, x, y, width_x, width_y)
 the gaussian parameters of a 2D distribution by calculating its
 moments """
    total = data.sum()
    X, Y = np.indices(data.shape)
    x = (X*data).sum()/total
    y = (Y*data).sum()/total
    col = data[:, int(y)]
    width_x = np.sqrt(np.abs((np.arange(col.size)-y)**2*col).sum()/
    row = data[int(x), :]
    width_y = np.sqrt(np.abs((np.arange(row.size)-x)**2*row).sum()/
    height = data.max()
    return height, x, y, width_x, width_y

def fitgaussian(data):
    """Returns (height, x, y, width_x, width_y)
 the gaussian parameters of a 2D distribution found by a fit"""
    params = moments(data)
    errorfunction = lambda p: np.ravel(gaussian(*p)(*np.indices(dat
                                 data)
    p, success = optimize.leastsq(errorfunction, params)
    return p
```

And here is an example using it:

```
# Create the gaussian data
Xin, Yin = np.mgrid[0:201, 0:201]
data = gaussian(3, 100, 100, 20, 40)(Xin, Yin) + np.random.random(X

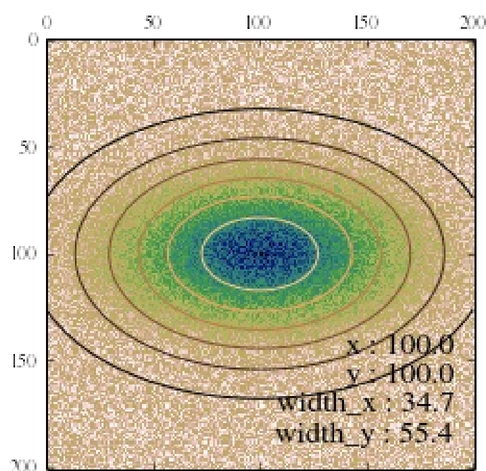
plt.matshow(data, cmap=plt.cm.gist_earth_r)

params = fitgaussian(data)
fit = gaussian(*params)

plt.contour(fit(*np.indices(data.shape)), cmap=plt.cm.copper)
ax = plt.gca()
(height, x, y, width_x, width_y) = params

plt.text(0.95, 0.05, """
x : %.1f
y : %.1f
width_x : %.1f
width_y : %.1f""" %(x, y, width_x, width_y),
        fontsize=16, horizontalalignment='right',
        verticalalignment='bottom', transform=ax.transAxes)
```

```
<matplotlib.text.Text at 0x7fab9d8a4dd0>
```



Fitting a power-law to data with errors

Generating the data

Generate some data with noise to demonstrate the fitting procedure. Data is generated with an amplitude of 10 and a power-law index of -2.0. Notice that all of our data is well-behaved when the log is taken... you may have to be more careful of this for real data.

```
# Define function for calculating a power law
powerlaw = lambda x, amp, index: amp * (x**index)

##########
# Generate data points with noise
##########
num_points = 20

# Note: all positive, non-zero data
xdata = np.linspace(1.1, 10.1, num_points)
ydata = powerlaw(xdata, 10.0, -2.0)     # simulated perfect data
yerr = 0.2 * ydata                      # simulated errors (10%)

ydata += np.random.randn(num_points) * yerr       # simulated nois
```

Fitting the data

If your data is well-behaved, you can fit a power-law function by first converting to a linear equation by using the logarithm. Then use the optimize function to fit a straight line. Notice that we are weighting by positional uncertainties during the fit. Also, the best-fit parameters uncertainties are estimated from the variance-covariance matrix. You should read up on when it may not be appropriate to use this form of error estimation. If you are trying to fit a power-law distribution, this solution is more appropriate.

```
##########
# Fitting the data -- Least Squares Method
##########

# Power-law fitting is best done by first converting
# to a linear equation and then fitting to a straight line.
#
#  y = a * x^b
#  log(y) = log(a) + b*log(x)
#

logx = np.log10(xdata)
logy = np.log10(ydata)
logyerr = yerr / ydata

# define our (line) fitting function
fitfunc = lambda p, x: p[0] + p[1] * x
errfunc = lambda p, x, y, err: (y - fitfunc(p, x)) / err

pinit = [1.0, -1.0]
out = optimize.leastsq(errfunc, pinit,
                       args=(logx, logy, logyerr), full_output=1)
```

```
pfinal = out[0]
covar = out[1]
print pfinal
print covar

index = pfinal[1]
amp = 10.0**pfinal[0]

indexErr = np.sqrt( covar[0][0] )
ampErr = np.sqrt( covar[1][1] ) * amp

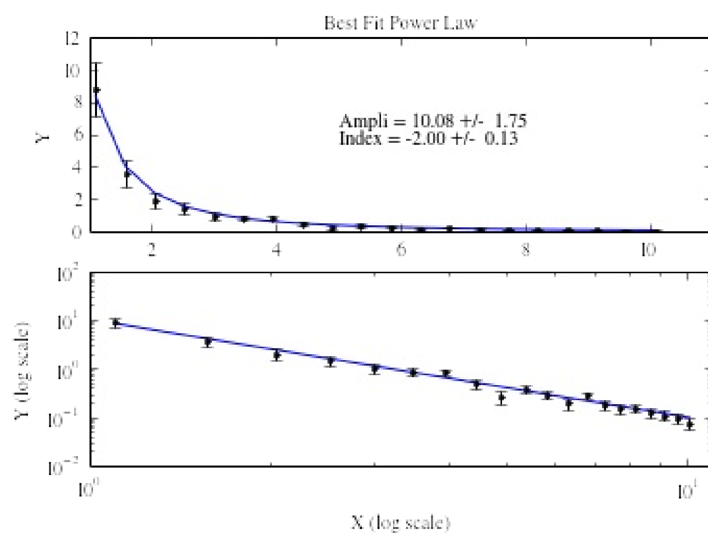
##########
# Plotting data
##########

plt.clf()
plt.subplot(2, 1, 1)
plt.plot(xdata, powerlaw(xdata, amp, index))     # Fit
plt.errorbar(xdata, ydata, yerr=yerr, fmt='k.')  # Data
plt.text(5, 6.5, 'Ampli = %5.2f +/- %5.2f' % (amp, ampErr))
plt.text(5, 5.5, 'Index = %5.2f +/- %5.2f' % (index, indexErr))
plt.title('Best Fit Power Law')
plt.xlabel('X')
plt.ylabel('Y')
plt.xlim(1, 11)

plt.subplot(2, 1, 2)
plt.loglog(xdata, powerlaw(xdata, amp, index))
plt.errorbar(xdata, ydata, yerr=yerr, fmt='k.')  # Data
plt.xlabel('X (log scale)')
plt.ylabel('Y (log scale)')
plt.xlim(1.0, 11)
```

```
[ 1.00341313 -2.00447676]
[[ 0.01592265 -0.0204523 ]
 [-0.0204523   0.03027352]]
```

```
(1.0, 11)
```



Attachments

- `gaussfitter.py`
- `gaussfitter2.py`

Large-scale bundle adjustment in scipy

A bundle adjusmtent problem arises in 3-D reconstruction and it can be formulated as follows (taken from https://en.wikipedia.org/wiki/Bundle_adjustment):

> Given a set of images depicting a number of 3D points from different viewpoints, bundle adjustment can be defined as the problem of simultaneously refining the 3D coordinates describing the scene geometry as well as the parameters of the relative motion and the optical characteristics of the camera(s) employed to acquire the images, according to an optimality criterion involving the corresponding image projections of all points.

More precisely. We have a set of points in real world defined by their coordinates `\((X, Y, Z)\)` in some apriori chosen "world coordinate frame". We photograph these points by different cameras, which are characterized by their orientation and translation relative to the world coordinate frame and also by focal length and two radial distortion parameters (9 parameters in total). Then we precicely measure 2-D coordinates `\((x, y)\)` of the points projected by the cameras on images. Our task is to refine 3-D coordinates of original points as well as camera parameters, by minimizing the sum of squares of reprojecting errors.

Let `\(\pmb{P} = (X, Y, Z)^T\)` - a radius-vector of a point, `\(\pmb{R}\)` - a rotation matrix of a camera, `\(\pmb{t}\)` - a translation vector of a camera, `\(f\)` - its focal distance, `\(k_1, k_2\)` - its distortion parameters. Then the reprojecting is done as follows:

```
\[\begin{align} \pmb{Q} = \pmb{R} \pmb{P} + \pmb{t} \\ \pmb{q} = -\
```

The resulting vector `\(\pmb{p}=(x, y)^T\)` contains image coordinates of the original point. This model is called "pinhole camera model", a very good notes about this subject I found here http://www.comp.nus.edu.sg/~cs4243/lecture/camera.pdf

---

Now let's start solving some real bundle adjusment problem. We'll take a problem from http://grail.cs.washington.edu/projects/bal/.

```
from __future__ import print_function
```

```
import urllib
import bz2
import os
import numpy as np
```

First download the data file:

```
BASE_URL = "http://grail.cs.washington.edu/projects/bal/data/ladybu
FILE_NAME = "problem-49-7776-pre.txt.bz2"
URL = BASE_URL + FILE_NAME
```

```
if not os.path.isfile(FILE_NAME):
    urllib.request.urlretrieve(URL, FILE_NAME)
```

Now read the data from the file:

```
def read_bal_data(file_name):
    with bz2.open(file_name, "rt") as file:
        n_cameras, n_points, n_observations = map(
            int, file.readline().split())

        camera_indices = np.empty(n_observations, dtype=int)
        point_indices = np.empty(n_observations, dtype=int)
        points_2d = np.empty((n_observations, 2))

        for i in range(n_observations):
            camera_index, point_index, x, y = file.readline().split
            camera_indices[i] = int(camera_index)
            point_indices[i] = int(point_index)
            points_2d[i] = [float(x), float(y)]

        camera_params = np.empty(n_cameras * 9)
        for i in range(n_cameras * 9):
            camera_params[i] = float(file.readline())
        camera_params = camera_params.reshape((n_cameras, -1))

        points_3d = np.empty(n_points * 3)
        for i in range(n_points * 3):
            points_3d[i] = float(file.readline())
        points_3d = points_3d.reshape((n_points, -1))

    return camera_params, points_3d, camera_indices, point_indices,
```

```
camera_params, points_3d, camera_indices, point_indices, points_2d
```

Here we have numpy arrays:

1. `camera_params` with shape `(n_cameras, 9)` contains initial estimates of parameters for all cameras. First 3 components in each row form a rotation vector (https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula), next 3 components form a translation vector, then a focal distance and two distortion parameters.
2. `points_3d` with shape `(n_points, 3)` contains initial estimates of point coordinates in the world frame.
3. `camera_ind` with shape `(n_observations,)` contains indices of cameras (from 0 to `n_cameras - 1`) involved in each observation.
4. `point_ind` with shape `(n_observations,)` contatins indices of points (from 0 to `n_points - 1`) involved in each observation.
5. `points_2d` with shape `(n_observations, 2)` contains measured 2-D coordinates of points projected on images in each observations.

And the numbers are:

```
n_cameras = camera_params.shape[0]
n_points = points_3d.shape[0]

n = 9 * n_cameras + 3 * n_points
m = 2 * points_2d.shape[0]

print("n_cameras: {}".format(n_cameras))
print("n_points: {}".format(n_points))
print("Total number of parameters: {}".format(n))
print("Total number of residuals: {}".format(m))
```

```
n_cameras: 49
n_points: 7776
Total number of parameters: 23769
Total number of residuals: 63686
```

We chose a relatively small problem to reduce computation time, but scipy's algorithm is capable of solving much larger problems, although required time will grow proportionally.

Now define the function which returns a vector of residuals. We use numpy vectorized computations:

```
def rotate(points, rot_vecs):
    """Rotate points by given rotation vectors.

 Rodrigues' rotation formula is used.
 """
    theta = np.linalg.norm(rot_vecs, axis=1)[:, np.newaxis]
    with np.errstate(invalid='ignore'):
        v = rot_vecs / theta
        v = np.nan_to_num(v)
    dot = np.sum(points * v, axis=1)[:, np.newaxis]
    cos_theta = np.cos(theta)
    sin_theta = np.sin(theta)

    return cos_theta * points + sin_theta * np.cross(v, points) + (
```

```
def project(points, camera_params):
    """Convert 3-D points to 2-D by projecting onto images."""
    points_proj = rotate(points, camera_params[:, :3])
    points_proj += camera_params[:, 3:6]
    points_proj = -points_proj[:, :2] / points_proj[:, 2, np.newaxi
    f = camera_params[:, 6]
    k1 = camera_params[:, 7]
    k2 = camera_params[:, 8]
    n = np.sum(points_proj**2, axis=1)
    r = 1 + k1 * n + k2 * n**2
    points_proj *= (r * f)[:, np.newaxis]
    return points_proj
```

```
def fun(params, n_cameras, n_points, camera_indices, point_indices,
    """Compute residuals.

 `params` contains camera parameters and 3-D coordinates.
 """
    camera_params = params[:n_cameras * 9].reshape((n_cameras, 9))
    points_3d = params[n_cameras * 9:].reshape((n_points, 3))
    points_proj = project(points_3d[point_indices], camera_params[c
    return (points_proj - points_2d).ravel()
```

You can see that computing Jacobian of `fun` is cumbersome, thus we will rely on the finite difference approximation. To make this process time feasible we provide Jacobian sparsity structure (i. e. mark elements which are known to be non-zero):

```
from scipy.sparse import lil_matrix
```

```
def bundle_adjustment_sparsity(n_cameras, n_points, camera_indices,
    m = camera_indices.size * 2
    n = n_cameras * 9 + n_points * 3
    A = lil_matrix((m, n), dtype=int)

    i = np.arange(camera_indices.size)
    for s in range(9):
        A[2 * i, camera_indices * 9 + s] = 1
        A[2 * i + 1, camera_indices * 9 + s] = 1

    for s in range(3):
        A[2 * i, n_cameras * 9 + point_indices * 3 + s] = 1
        A[2 * i + 1, n_cameras * 9 + point_indices * 3 + s] = 1

    return A
```

Now we are ready to run optimization. Let’s visualize residuals evaluated with the initial parameters.

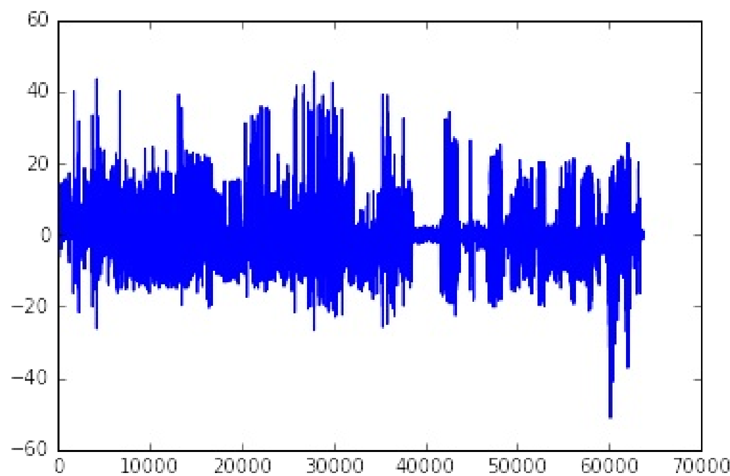
```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
x0 = np.hstack((camera_params.ravel(), points_3d.ravel()))
```

```
f0 = fun(x0, n_cameras, n_points, camera_indices, point_indices, po
```

```
plt.plot(f0)
```

```
[<matplotlib.lines.Line2D at 0x1067696d8>]
```




```
A = bundle_adjustment_sparsity(n_cameras, n_points, camera_indices,
```

```
import time
from scipy.optimize import least_squares
```

```
t0 = time.time()
res = least_squares(fun, x0, jac_sparsity=A, verbose=2, x_scale='ja
                    args=(n_cameras, n_points, camera_indices, poir
t1 = time.time()
```

```
   Iteration     Total nfev        Cost      Cost reduction    Ste
       0              1         8.5091e+05
       1              3         5.0985e+04      8.00e+05       1.4
       2              4         1.6077e+04      3.49e+04       2.5
       3              5         1.4163e+04      1.91e+03       2.8
       4              7         1.3695e+04      4.67e+02       1.3
       5              8         1.3481e+04      2.14e+02       2.2
       6              9         1.3436e+04      4.55e+01       3.1
       7             10         1.3422e+04      1.37e+01       6.8
       8             11         1.3418e+04      3.70e+00       1.2
       9             12         1.3414e+04      4.19e+00       2.6
      10             13         1.3412e+04      1.88e+00       7.5
      11             14         1.3410e+04      2.09e+00       1.7
      12             15         1.3409e+04      1.04e+00       4.0
 ftol termination condition is satisfied.
 Function evaluations 15, initial cost 8.5091e+05, final cost 1.340
```

```
print("Optimization took {0:.0f} seconds".format(t1 - t0))
```

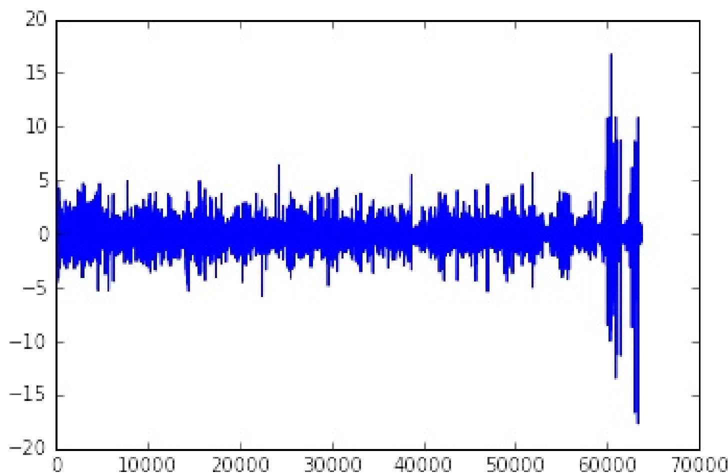
```
Optimization took 33 seconds
```

Setting `scaling='jac'` was done to automatically scale the variables and equalize their influence on the cost function (clearly the camera parameters and coordinates of the points are very different entities). This option turned out to be crucial for successfull bundle adjustment.

Now let’s plot residuals at the found solution:

```
plt.plot(res.fun)
```

```
[<matplotlib.lines.Line2D at 0x10de73438>]
```



We see much better picture of residuals now, with the mean being very close to zero. There are some spikes left. It can be explained by outliers in the data, or, possibly, the algorithm found a local minimum (very good one though) or didn’t converged enough. Note that the algorithm worked with Jacobian finite difference aproximate, which can potentially block the progress near the minimum because of insufficient accuracy (but again, computing exact Jacobian for this problem is quite difficult).

Least squares circle

Introduction

This page gathers different methods used to find the least squares circle fitting a set of 2D points (x,y).

The full code of this analysis is available here: `least_squares_circle_v1d.py`.

Finding the least squares circle corresponds to finding the center of the circle (xc, yc) and its radius Rc which minimize the residu function defined below:

```
#! python
Ri = sqrt( (x - xc)**2 + (y - yc)**2)
residu = sum( (Ri - Rc)**2)
```

This is a nonlinear problem. We well see three approaches to the problem, and compare there results, as well as their speeds.

Using an algebraic approximation

As detailed in this document this problem can be approximated by a linear one if we define the function to minimize as follow:

```
#! python
residu_2 = sum( (Ri**2 - Rc**2)**2)
```

This leads to the following method, using linalg.solve :

```
#! python
# == METHOD 1 ==
method_1 = 'algebraic'

# coordinates of the barycenter
x_m = mean(x)
y_m = mean(y)

# calculation of the reduced coordinates
u = x - x_m
v = y - y_m

# linear system defining the center (uc, vc) in reduced coordinates
#    Suu * uc +  Suv * vc = (Suuu + Suvv)/2
#    Suv * uc +  Svv * vc = (Suuv + Svvv)/2
Suv  = sum(u*v)
Suu  = sum(u**2)
Svv  = sum(v**2)
Suuv = sum(u**2 * v)
Suvv = sum(u * v**2)
Suuu = sum(u**3)
Svvv = sum(v**3)

# Solving the linear system
A = array([ [ Suu, Suv ], [Suv, Svv]])
B = array([ Suuu + Suvv, Svvv + Suuv ])/2.0
uc, vc = linalg.solve(A, B)

xc_1 = x_m + uc
yc_1 = y_m + vc

# Calcul des distances au centre (xc_1, yc_1)
Ri_1     = sqrt((x-xc_1)**2 + (y-yc_1)**2)
R_1      = mean(Ri_1)
residu_1 = sum((Ri_1-R_1)**2)
```

Using scipy.optimize.leastsq

Scipy comes will several tools to solve the nonlinear problem above. Among them, scipy.optimize.leastsq is very simple to use in this case.

Indeed, once the center of the circle is defined, the radius can be calculated directly and is equal to mean(Ri). So there is only two parameters left: xc and yc.

Basic usage

```
#! python
#  == METHOD 2 ==
from scipy      import optimize

method_2 = "leastsq"

def calc_R(xc, yc):
    """ calculate the distance of each 2D points from the center (x
    return sqrt((x-xc)**2 + (y-yc)**2)

def f_2(c):
    """ calculate the algebraic distance between the data points an
    Ri = calc_R(*c)
    return Ri - Ri.mean()

center_estimate = x_m, y_m
center_2, ier = optimize.leastsq(f_2, center_estimate)

xc_2, yc_2 = center_2
Ri_2       = calc_R(*center_2)
R_2        = Ri_2.mean()
residu_2   = sum((Ri_2 - R_2)**2)
```

Advanced usage, with jacobian function

To gain in speed, it is possible to tell optimize.leastsq how to compute the jacobian of the function by adding a Dfun argument:

```
#! python
# == METHOD 2b ==
method_2b  = "leastsq with jacobian"

def calc_R(xc, yc):
    """ calculate the distance of each data points from the center
    return sqrt((x-xc)**2 + (y-yc)**2)

def f_2b(c):
    """ calculate the algebraic distance between the 2D points and
    Ri = calc_R(*c)
    return Ri - Ri.mean()

def Df_2b(c):
    """ Jacobian of f_2b
 The axis corresponding to derivatives must be coherent with the co
    xc, yc     = c
    df2b_dc    = empty((len(c), x.size))

    Ri = calc_R(xc, yc)
    df2b_dc[0] = (xc - x)/Ri                   # dR/dxc
    df2b_dc[1] = (yc - y)/Ri                   # dR/dyc
    df2b_dc    = df2b_dc - df2b_dc.mean(axis=1)[:, newaxis]

    return df2b_dc

center_estimate = x_m, y_m
center_2b, ier = optimize.leastsq(f_2b, center_estimate, Dfun=Df_2b

xc_2b, yc_2b = center_2b
Ri_2b        = calc_R(*center_2b)
R_2b         = Ri_2b.mean()
residu_2b    = sum((Ri_2b - R_2b)**2)
```

Using scipy.odr

Scipy has a dedicated package to deal with orthogonal distance regression, namely scipy.odr. This package can handle both explict and implicit function definition, and we will used the second form in this case.

Here is the implicit definition of the circle:

```
#! python
(x - xc)**2 + (y-yc)**2 - Rc**2 = 0
```

Basic usage

This leads to the following code:

```
#! python
# == METHOD 3 ==
from scipy      import  odr

method_3 = "odr"

def f_3(beta, x):
    """ implicit definition of the circle """
    return (x[0]-beta[0])**2 + (x[1]-beta[1])**2 -beta[2]**2

# initial guess for parameters
R_m = calc_R(x_m, y_m).mean()
beta0 = [ x_m, y_m, R_m]

# for implicit function :
#       data.x contains both coordinates of the points (data.x = [x
#       data.y is the dimensionality of the response
lsc_data  = odr.Data(row_stack([x, y]), y=1)
lsc_model = odr.Model(f_3, implicit=True)
lsc_odr   = odr.ODR(lsc_data, lsc_model, beta0)
lsc_out   = lsc_odr.run()

xc_3, yc_3, R_3 = lsc_out.beta
Ri_3 = calc_R([xc_3, yc_3])
residu_3 = sum((Ri_3 - R_3)**2)
```

Advanced usage, with jacobian functions

One of the advantages of the implicit function definition is that its derivatives are very easily calculated.

This can be used to complete the model:

```
#! python
# == METHOD 3b ==
method_3b  = "odr with jacobian"

def f_3b(beta, x):
    """ implicit definition of the circle """
    return (x[0]-beta[0])**2 + (x[1]-beta[1])**2 -beta[2]**2

def jacb(beta, x):
    """ Jacobian function with respect to the parameters beta.
 return df_3b/dbeta
 """
    xc, yc, r = beta
    xi, yi    = x
```

```
    df_db    = empty((beta.size, x.shape[1]))
    df_db[0] =  2*(xc-xi)                     # d_f/dxc
    df_db[1] =  2*(yc-yi)                     # d_f/dyc
    df_db[2] = -2*r                           # d_f/dr

    return df_db

def jacd(beta, x):
    """ Jacobian function with respect to the input x.
 return df_3b/dx
 """
    xc, yc, r = beta
    xi, yi    = x

    df_dx    = empty_like(x)
    df_dx[0] =  2*(xi-xc)                     # d_f/dxi
    df_dx[1] =  2*(yi-yc)                     # d_f/dyi

    return df_dx

def calc_estimate(data):
    """ Return a first estimation on the parameter from the data  '
    xc0, yc0 = data.x.mean(axis=1)
    r0 = sqrt((data.x[0]-xc0)**2 +(data.x[1] -yc0)**2).mean()
    return xc0, yc0, r0

# for implicit function :
#       data.x contains both coordinates of the points
#       data.y is the dimensionality of the response
lsc_data  = odr.Data(row_stack([x, y]), y=1)
lsc_model = odr.Model(f_3b, implicit=True, estimate=calc_estimate,
lsc_odr   = odr.ODR(lsc_data, lsc_model)    # beta0 has been replac
lsc_odr.set_job(deriv=3)                    # use user derivatives
lsc_odr.set_iprint(iter=1, iter_step=1)     # print details for eac
lsc_out   = lsc_odr.run()

xc_3b, yc_3b, R_3b = lsc_out.beta
Ri_3b       = calc_R(xc_3b, yc_3b)
residu_3b   = sum((Ri_3b - R_3b)**2)
```

Comparison of the three methods

We will compare the results of these three methods in two cases:

```
* when 2D points are all around the circle
```

```
* when 2D points are in a small arc
```

Data points all around the circle

Here is an example with data points all around the circle:

```
#! python
# Coordinates of the 2D points
x = r_[  9,  35, -13,  10,  23,   0]
y = r_[ 34,  10,   6, -14,  27, -10]
```

This gives:

||||||||||||||**SUMMARY**|| ||**Method**|| **Xc** || **Yc** || **Rc** ||**nb_calls** || **std(Ri)**|| **residu** || **residu2** || ||algebraic || 10.55231 || 9.70590|| 23.33482|| 1|| 1.135135|| 7.731195|| 16236.34|| ||leastsq || 10.50009 || 9.65995|| 23.33353|| 15|| 1.133715|| 7.711852|| 16276.89|| ||leastsq with jacobian || 10.50009 || 9.65995|| 23.33353|| 7|| 1.133715|| 7.711852|| 16276.89|| ||odr || 10.50009 || 9.65995|| 23.33353|| 82|| 1.133715|| 7.711852|| 16276.89|| ||odr with jacobian || 10.50009 || 9.65995|| 23.33353|| 16|| 1.133715|| 7.711852|| 16276.89||

Note:

`*` nb_calls `correspond to the number of function calls of the functior`

`* as shown on the figures below, the two functions` residu `and` residu_



Data points around an arc

Here is an example where data points form an arc:

```
#! python
x = r_[36, 36, 19, 18, 33, 26]
y = r_[14, 10, 28, 31, 18, 26]
```

'''Method'''	'''Xc'''	'''Yc'''	'''Rc'''	'''nb_calls'''	'''std(Ri)'''
algebraic	15.05503	8.83615	20.82995	1	0.930508
leastsq	9.88760	3.68753	27.35040	24	0.820825
leastsq with jacobian	9.88759	3.68752	27.35041	10	0.820825
odr	9.88757	3.68750	27.35044	472	0.820825
odr with jacobian	9.88757	3.68750	27.35044	109	0.820825

`arc_v5.png` `arc_residu2_v6.png`

Conclusion

ODR and leastsq gives the same results.

Optimize.leastsq is the most efficient method, and can be two to ten times faster than ODR, at least as regards the number of function call.

Adding a function to compute the jacobian can lead to decrease the number of function calls by a factor of two to five.

In this case, to use ODR seems a bit overkill but it can be very handy for more complex use cases like ellipses.

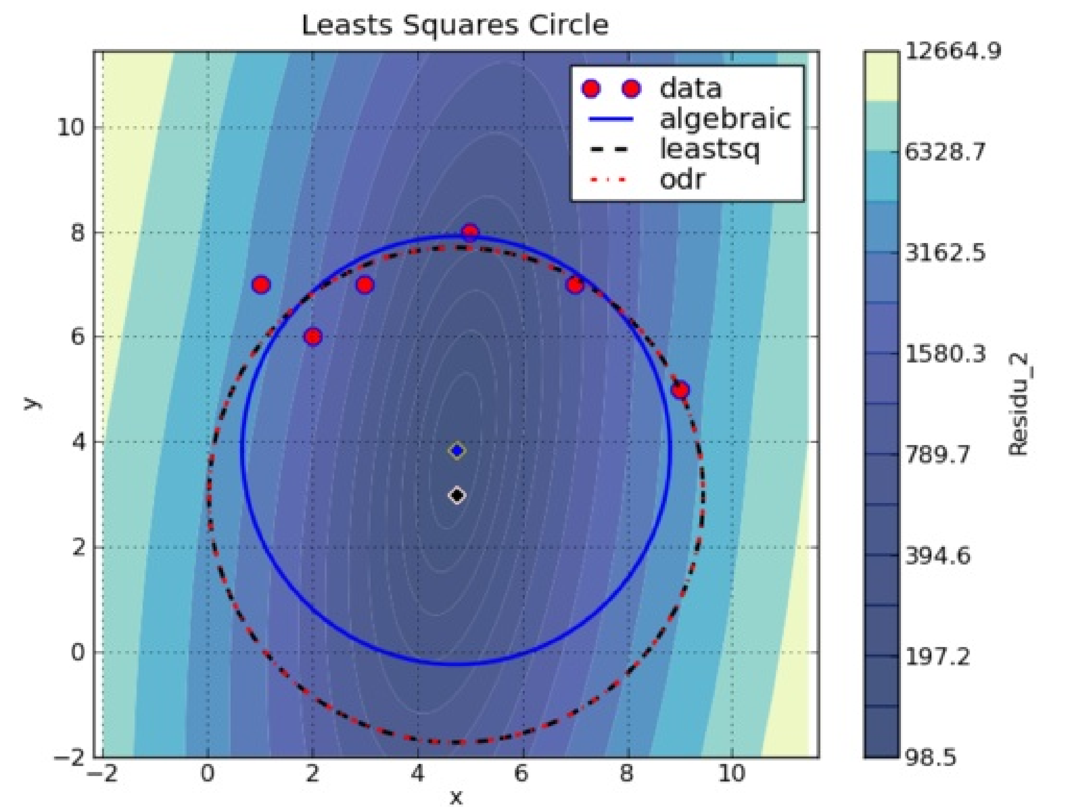
The algebraic approximation gives good results when the points are all around the circle but is limited when there is only an arc to fit.

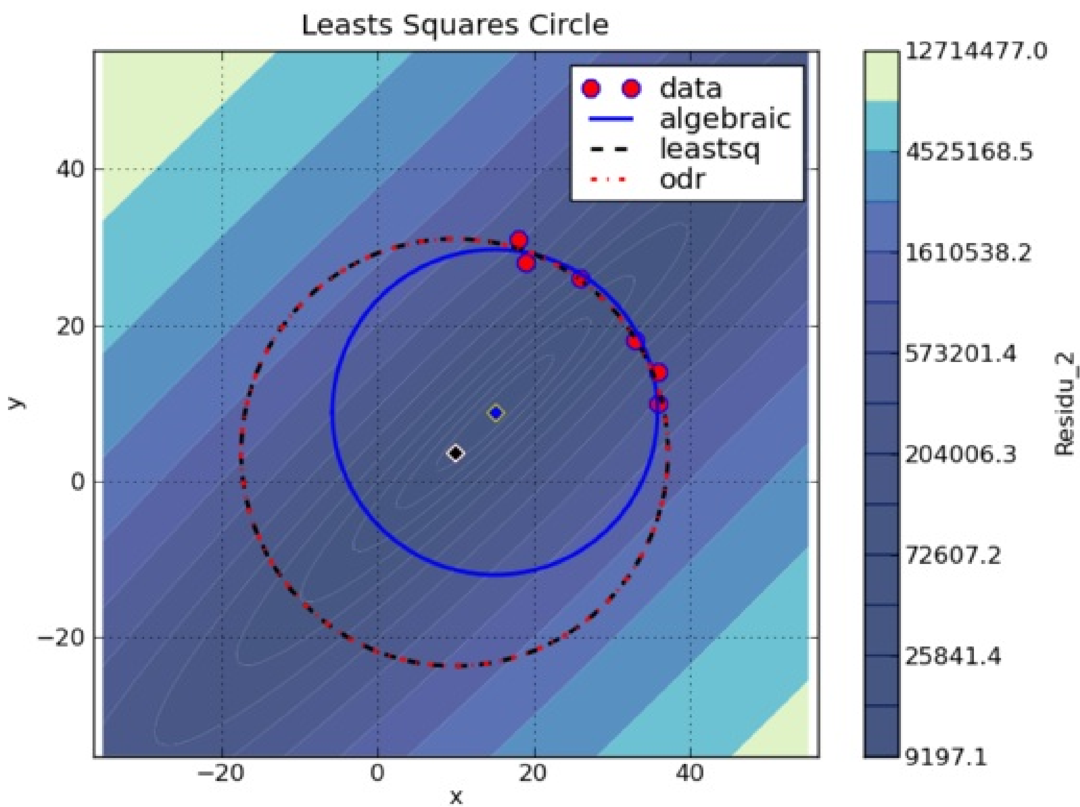
Indeed, the two errors functions to minimize are not equivalent when data points are not all exactly on a circle. The algebraic method leads in most of the case to a smaller radius than that of the least squares circle, as its error function is based on squared distances and not on the distance themselves.

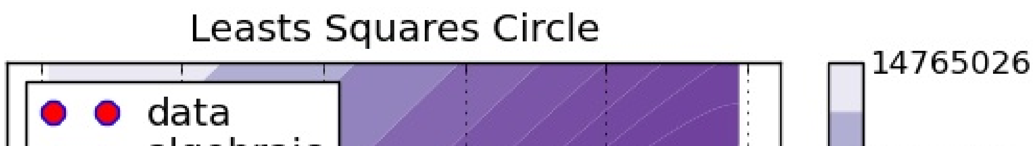
Attachments

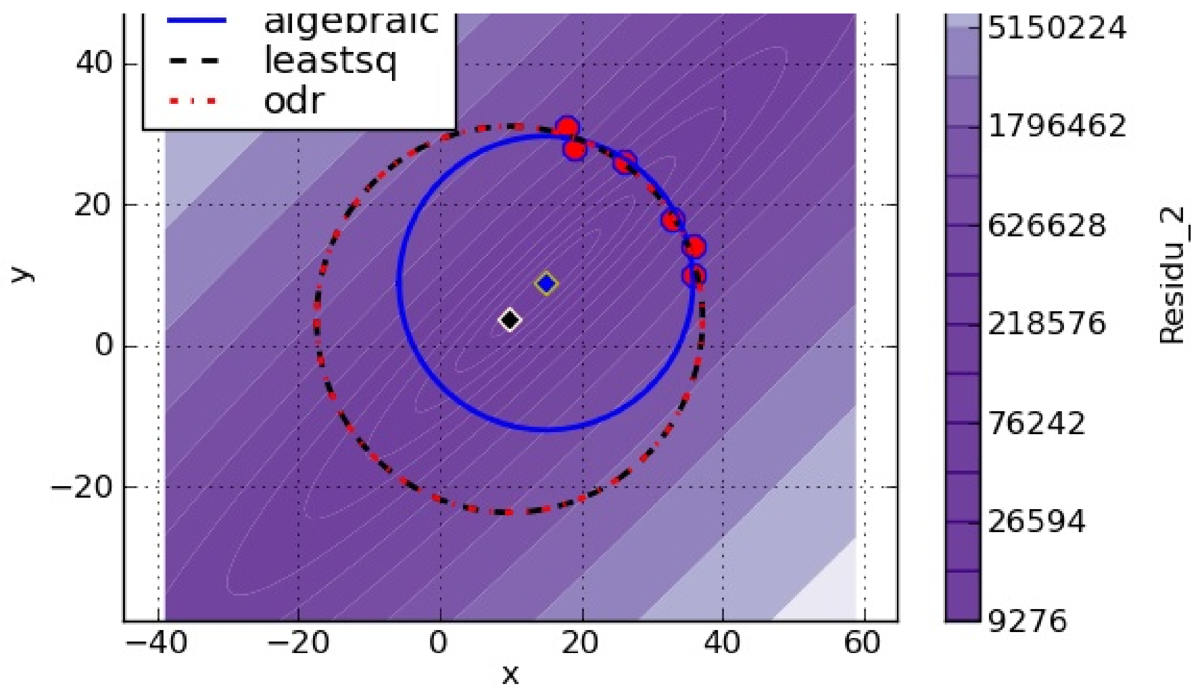
- `arc_residu2_v1.png`
- `arc_residu2_v2.png`
- `arc_residu2_v3.png`
- `arc_residu2_v4.png`
- `arc_residu2_v5.png`
- `arc_residu2_v6.png`
- `arc_v1.png`
- `arc_v2.png`
- `arc_v3.png`
- `arc_v4.png`
- `arc_v5.png`
- `full_cercle_residu2_v1.png`
- `full_cercle_residu2_v2.png`
- `full_cercle_residu2_v3.png`
- `full_cercle_residu2_v4.png`
- `full_cercle_residu2_v5.png`
- `full_cercle_v1.png`
- `full_cercle_v2.png`
- `full_cercle_v3.png`
- `full_cercle_v4.png`
- `full_cercle_v5.png`
- `least_squares_circle.py`

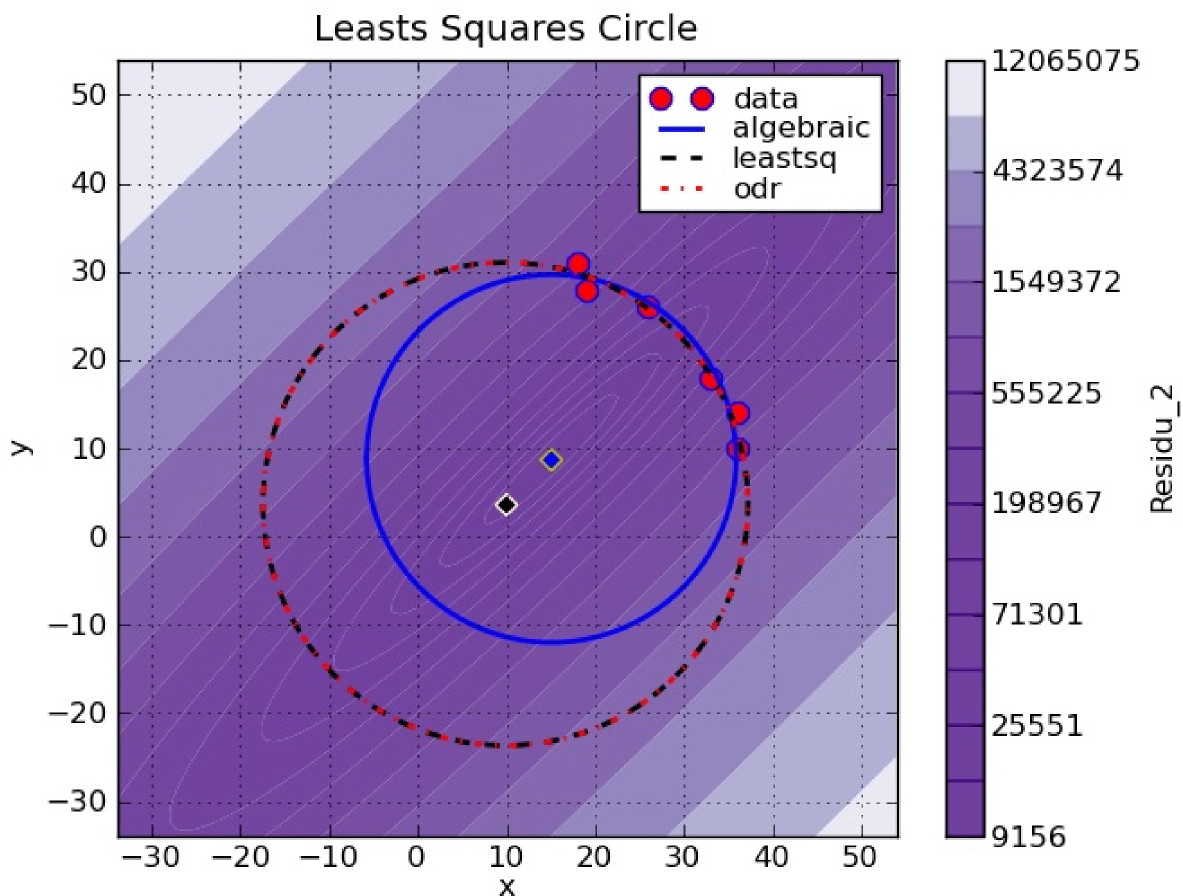
- `least_squares_circle_v1.py`
- `least_squares_circle_v1b.py`
- `least_squares_circle_v1c.py`
- `least_squares_circle_v1d.py`
- `least_squares_circle_v2.py`
- `least_squares_circle_v3.py`

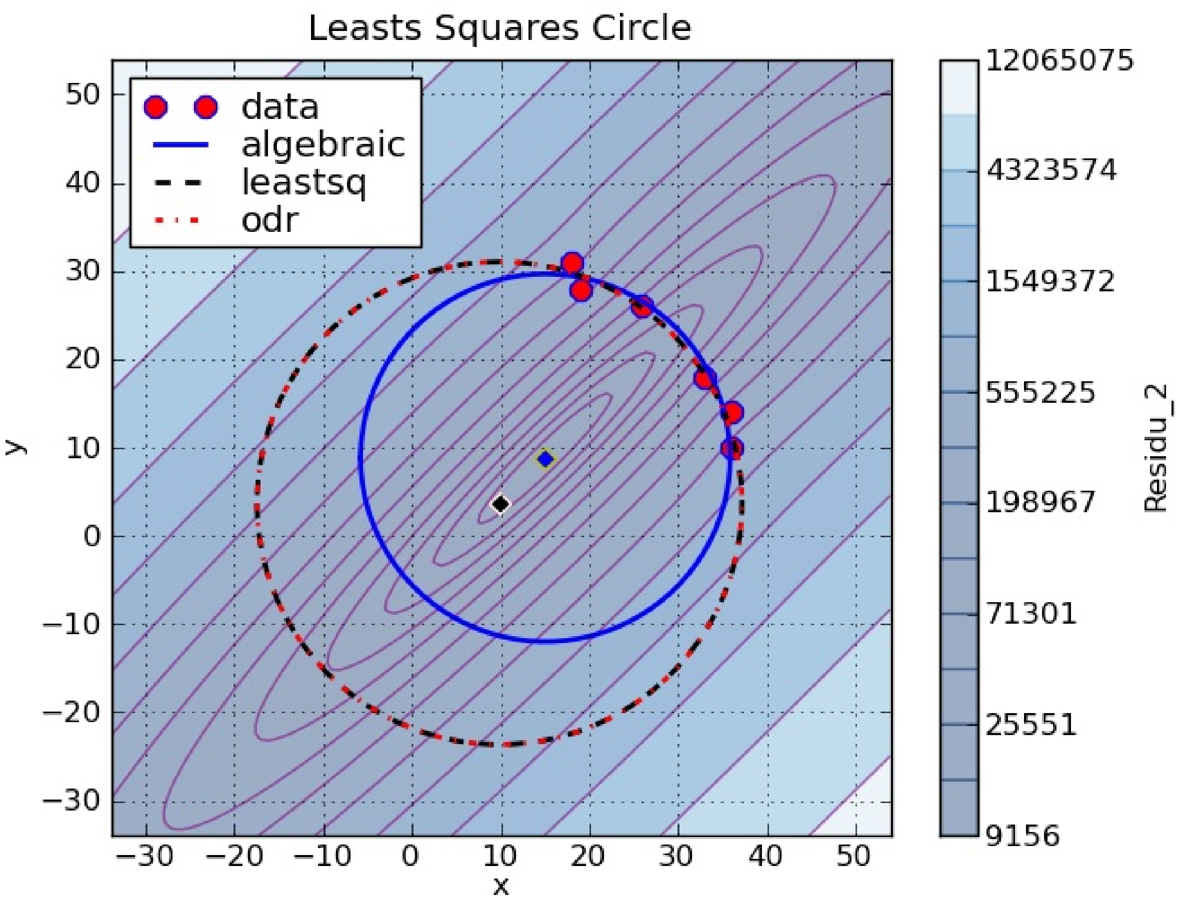

Leasts Squares Circle
data
algebraic
leastsq
odr
Residu_2
12664.9
6328.7
3162.5
1580.3
789.7
394.6
197.2
98.5
x
y

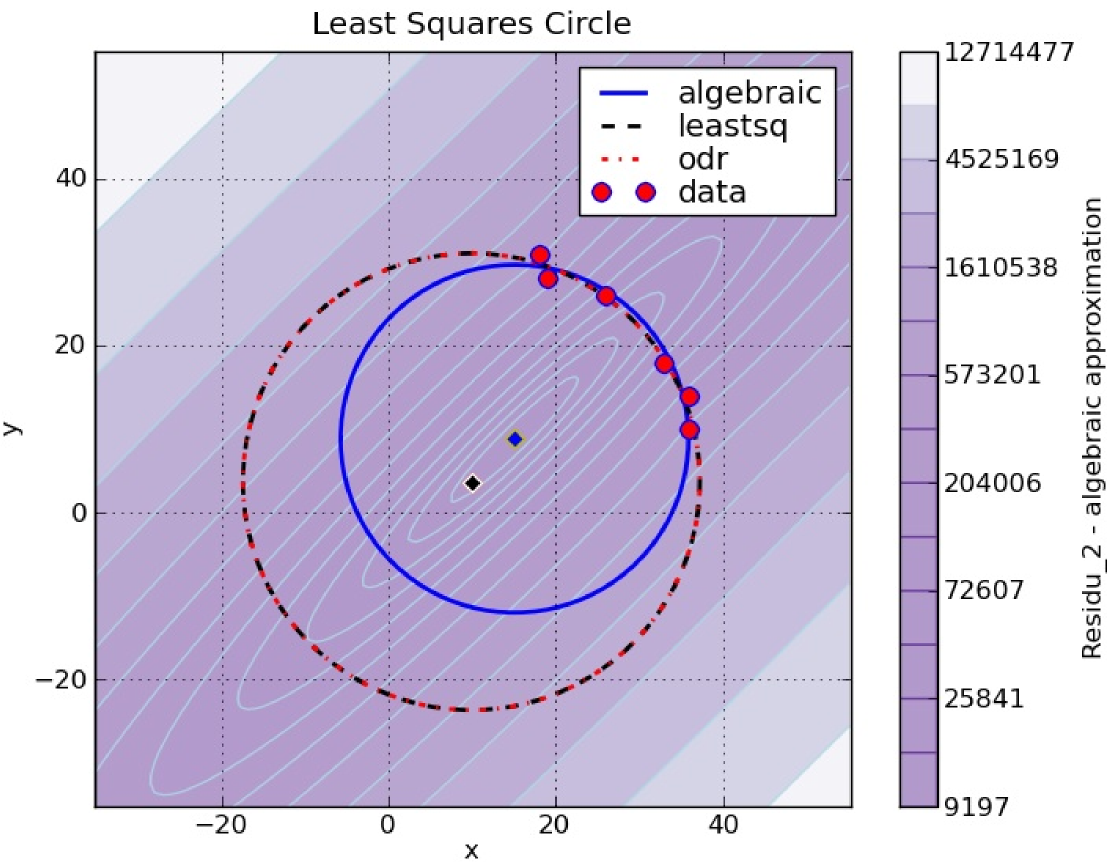

Leasts Squares Circle
data
algebraic
leastsq
odr
Residu_2
12714477.0
4525168.5
1610538.2
573201.4
204006.3
72607.2
25841.4
9197.1
x
y

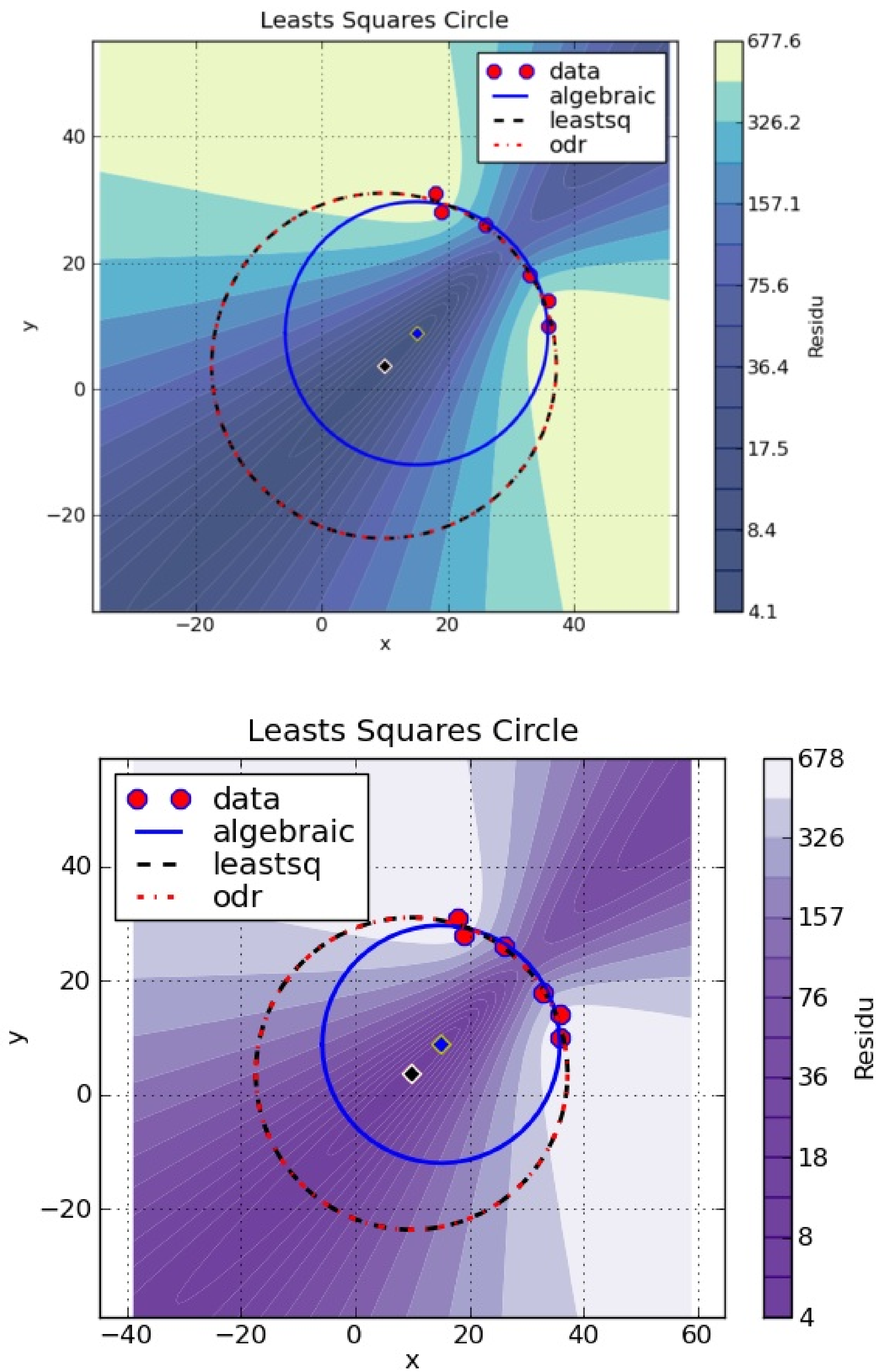

Leasts Squares Circle
data
14765026

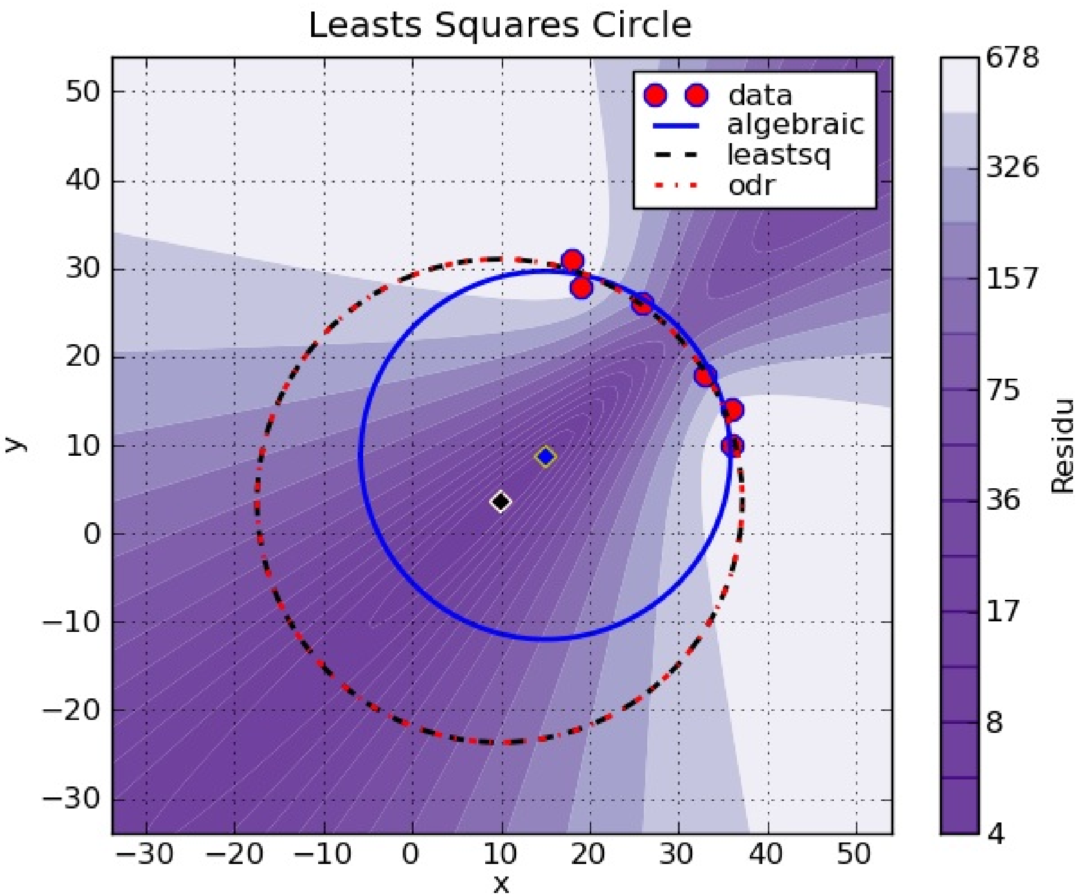

algebraic
leastsq
odr
40
20
0
−20
y
−40
−20
0
20
40
60
x
5150224
1796462
626628
218576
76242
26594
9276
Residu_2


Leasts Squares Circle
data
algebraic
leastsq
odr
50
40
30
20
10
0
−10
−20
−30
y
−30
−20
−10
0
10
20
30
40
50
x
12065075
4323574
1549372
555225
198967
71301
25551
9156
Residu_2

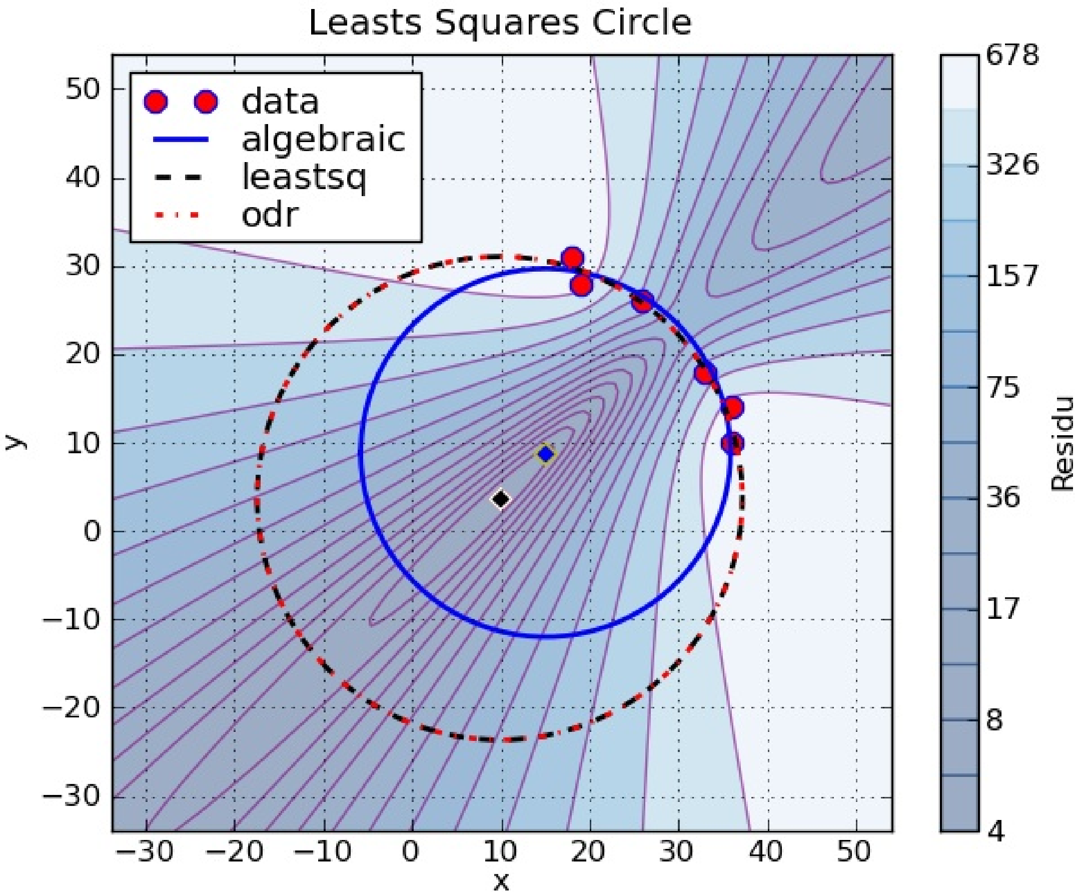

Leasts Squares Circle
data
algebraic
leastsq
odr
x
y
Residu_2
12065075
4323574
1549372
555225
198967
71301
25551
9156


Least Squares Circle
algebraic
leastsq
odr
data
x
y
Residu_2 - algebraic approximation
12714477
4525169
1610538
573201
204006
72607
25841
9197

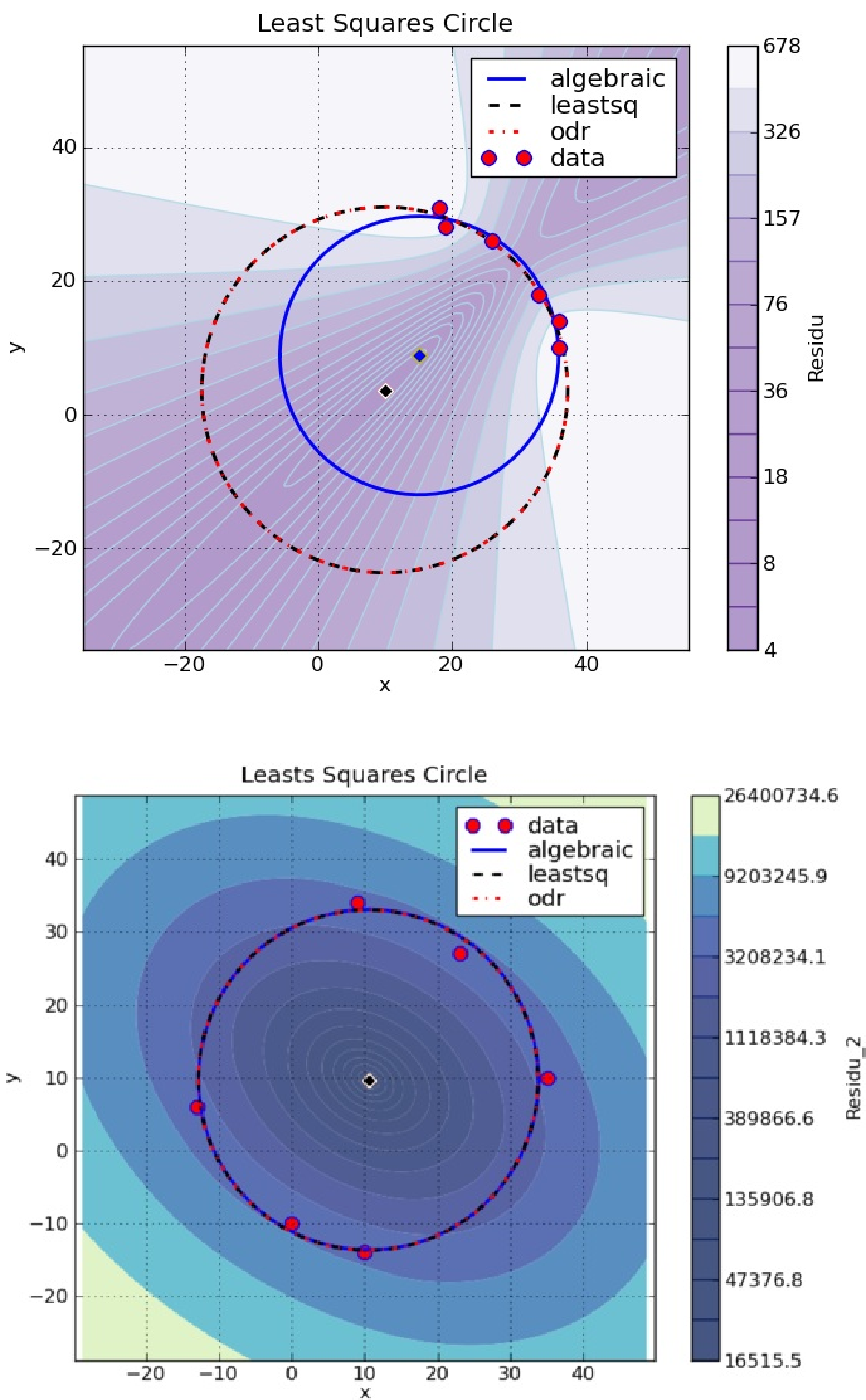

Leasts Squares Circle
data
algebraic
leastsq
odr
x
y
Residu
677.6
326.2
157.1
75.6
36.4
17.5
8.4
4.1
Leasts Squares Circle
data
algebraic
leastsq
odr
x
y
Residu
678
326
157
76
36
18
8
4

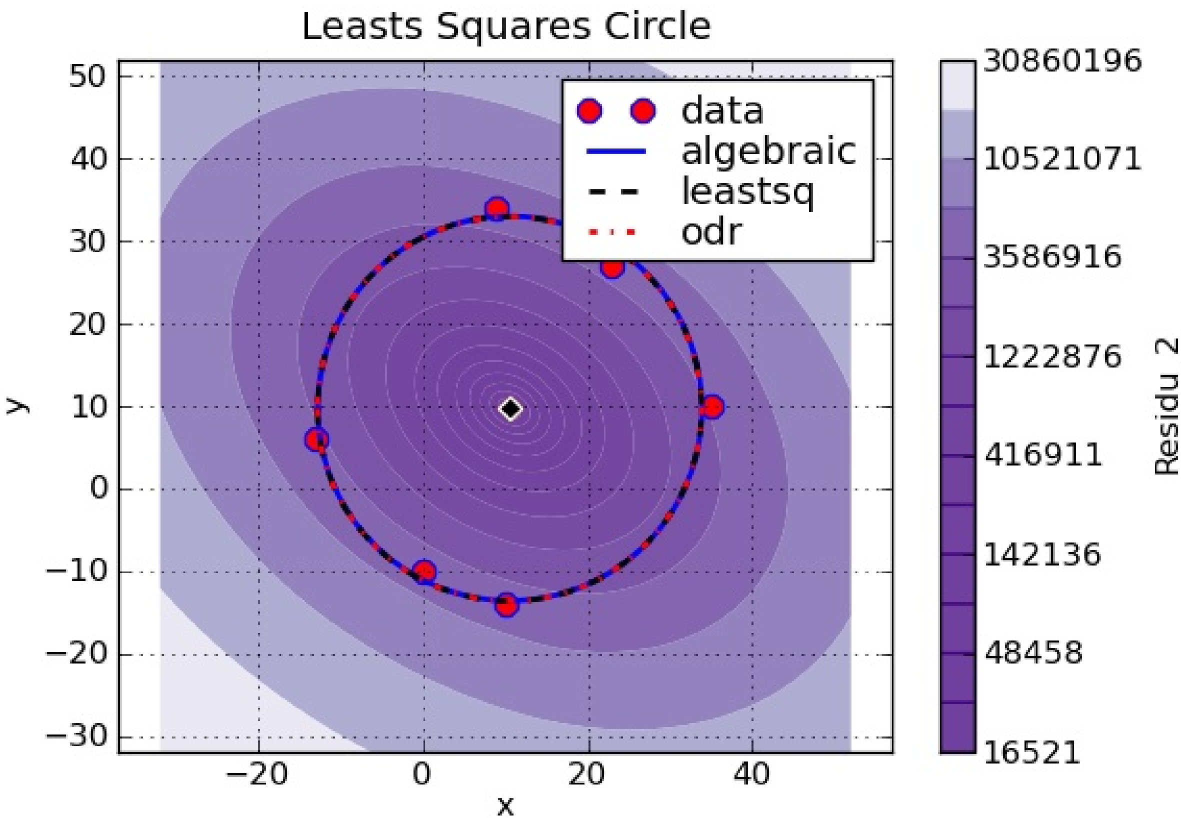


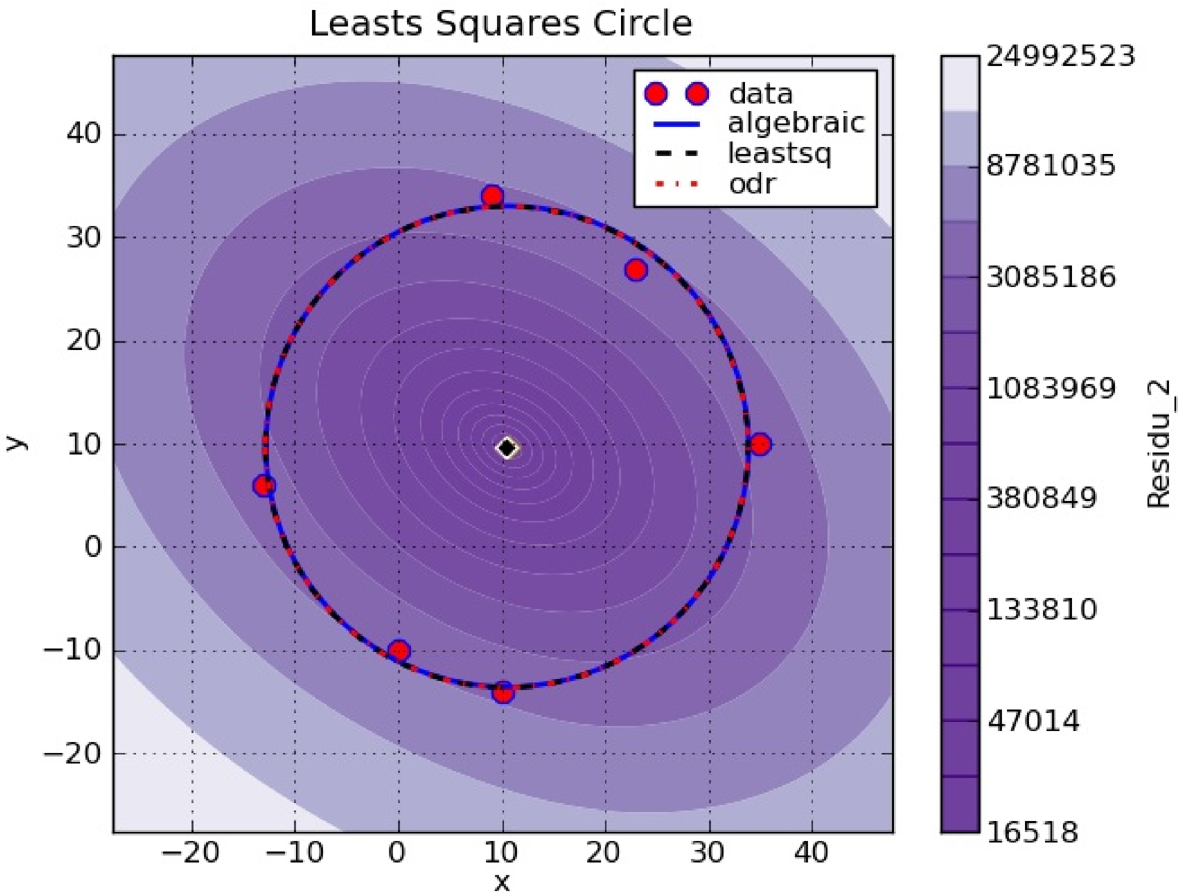
Leasts Squares Circle
data
algebraic
leastsq
odr
x
y
Residu
678
326
157
75
36
17
8
4

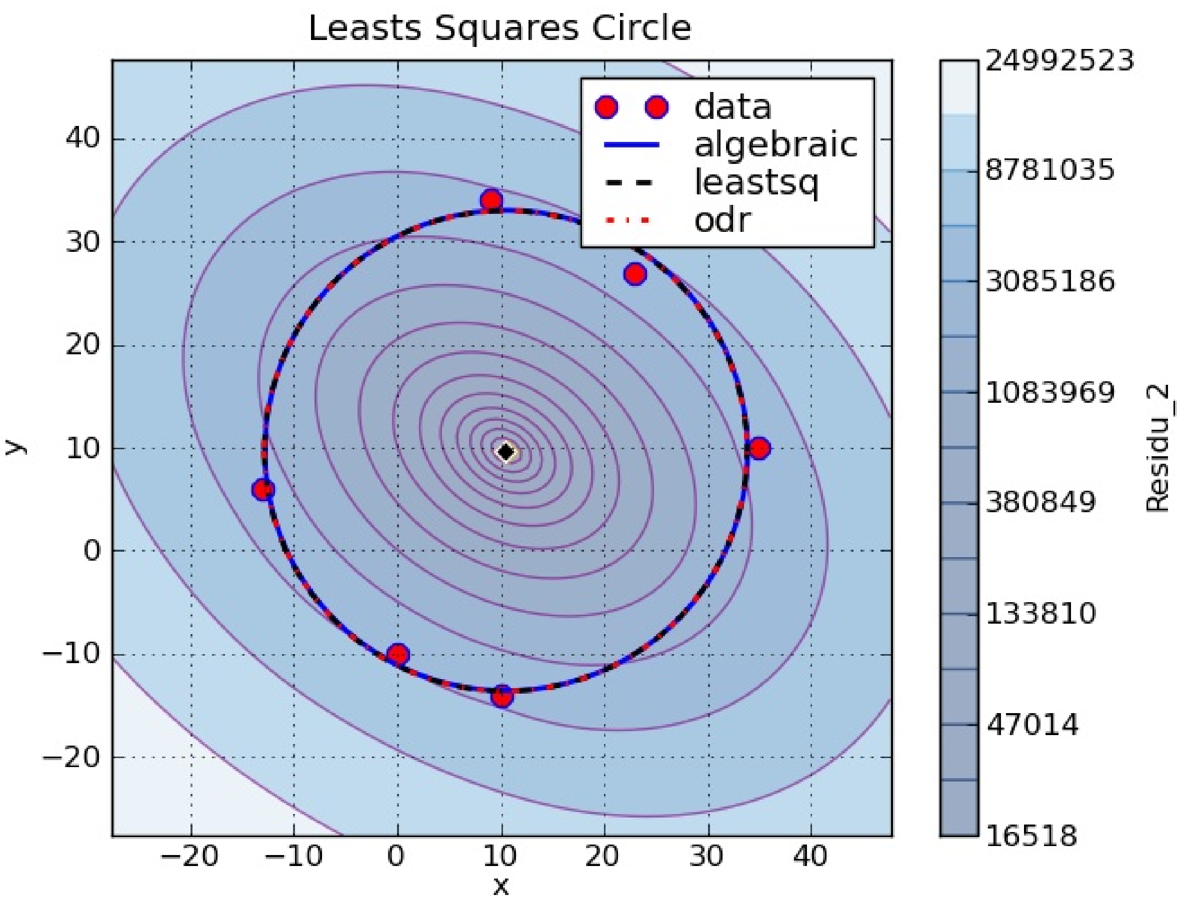


Leasts Squares Circle
data
algebraic
leastsq
odr
x
y
Residu
678
326
157
75
36
17
8
4

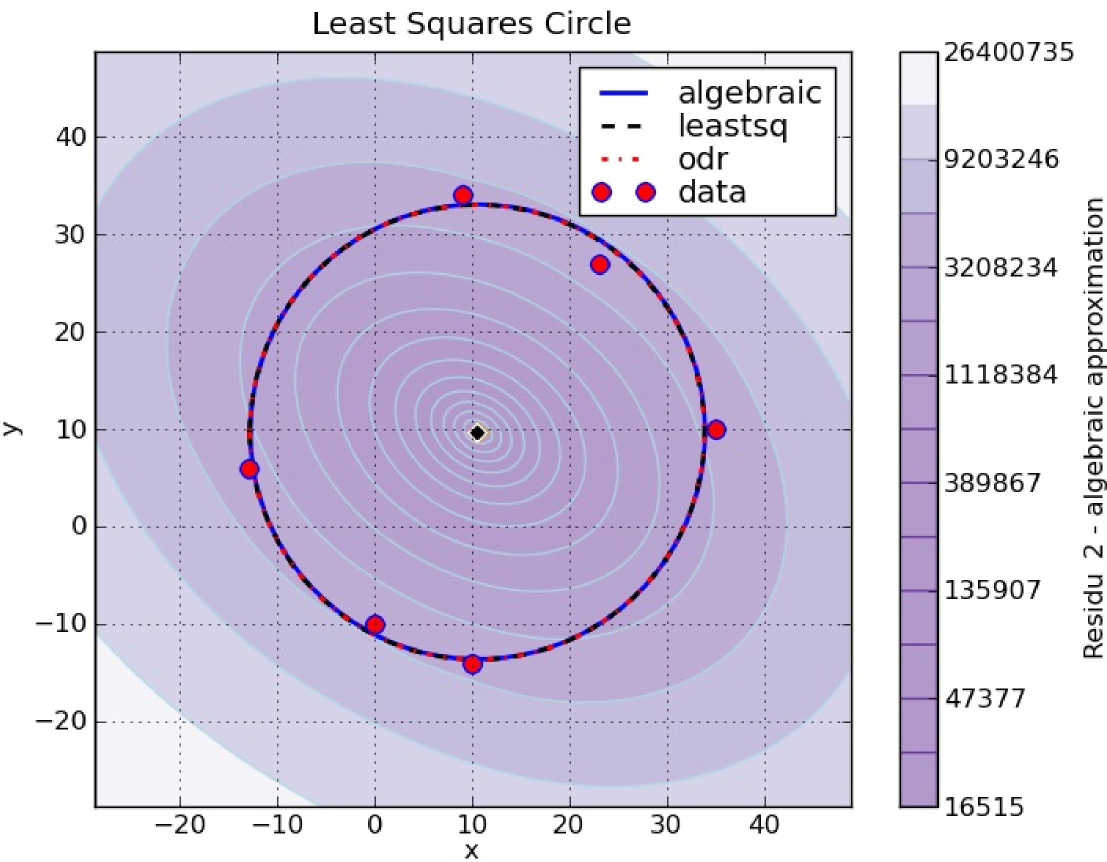

Least Squares Circle
algebraic
leastsq
odr
data
Residu
678
326
157
76
36
18
8
4
40
20
0
−20
−20
0
20
40
x
y
Leasts Squares Circle
data
algebraic
leastsq
odr
Residu_2
26400734.6
9203245.9
3208234.1
1118384.3
389866.6
135906.8
47376.8
16515.5
40
30
20
10
0
−10
−20
−20
−10
0
10
20
30
40
x
y


Leasts Squares Circle
data
algebraic
leastsq
odr
y
x
50
40
30
20
10
0
−10
−20
−30
−20
0
20
40
30860196
10521071
3586916
1222876
416911
142136
48458
16521
Residu_2

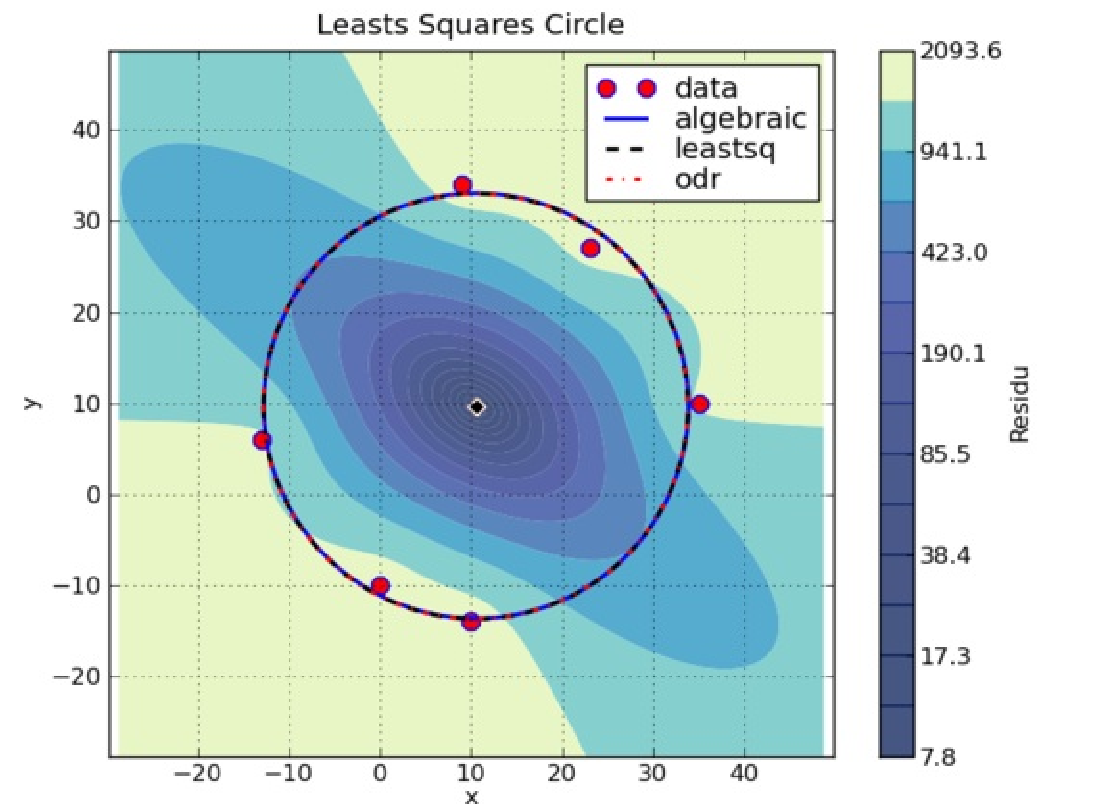

Leasts Squares Circle
data
algebraic
leastsq
odr
y
x
40
30
20
10
0
−10
−20
−20
−10
0
10
20
30
40
24992523
8781035
3085186
1083969
380849
133810
47014
16518
Residu_2

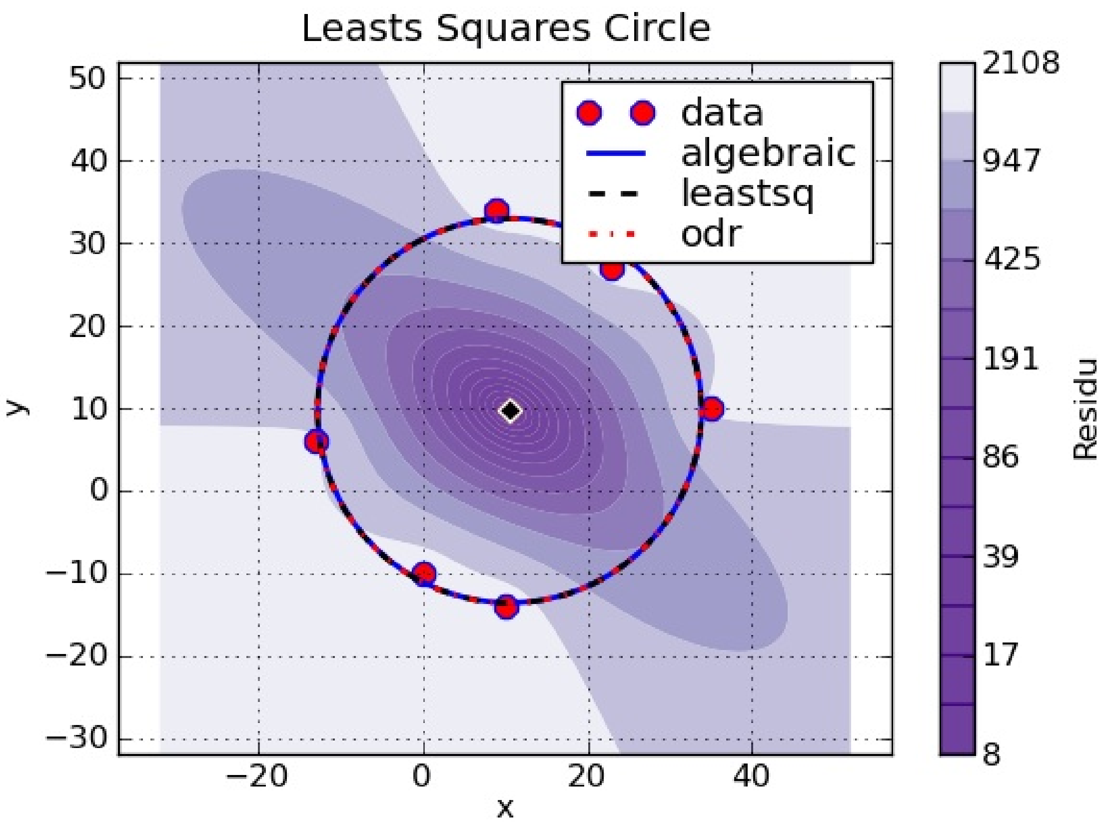


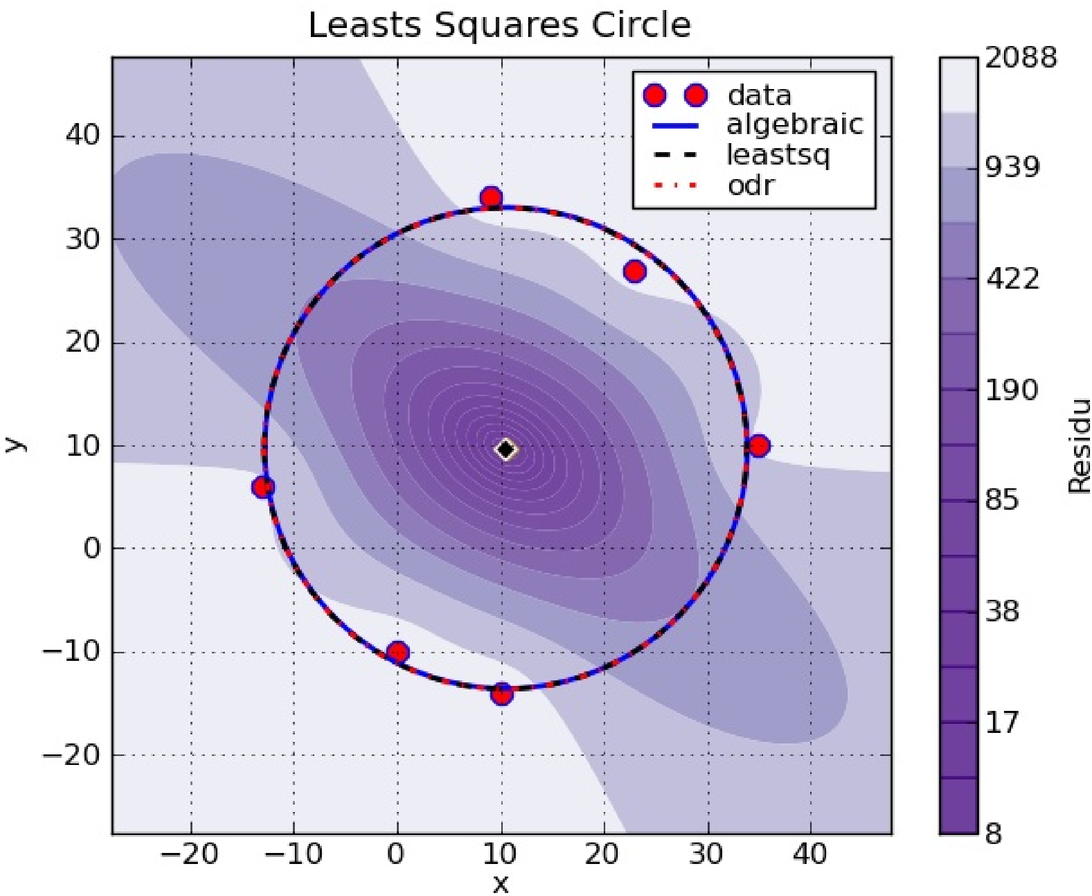
Leasts Squares Circle
data
algebraic
leastsq
odr
x
y
Residu_2
24992523
8781035
3085186
1083969
380849
133810
47014
16518

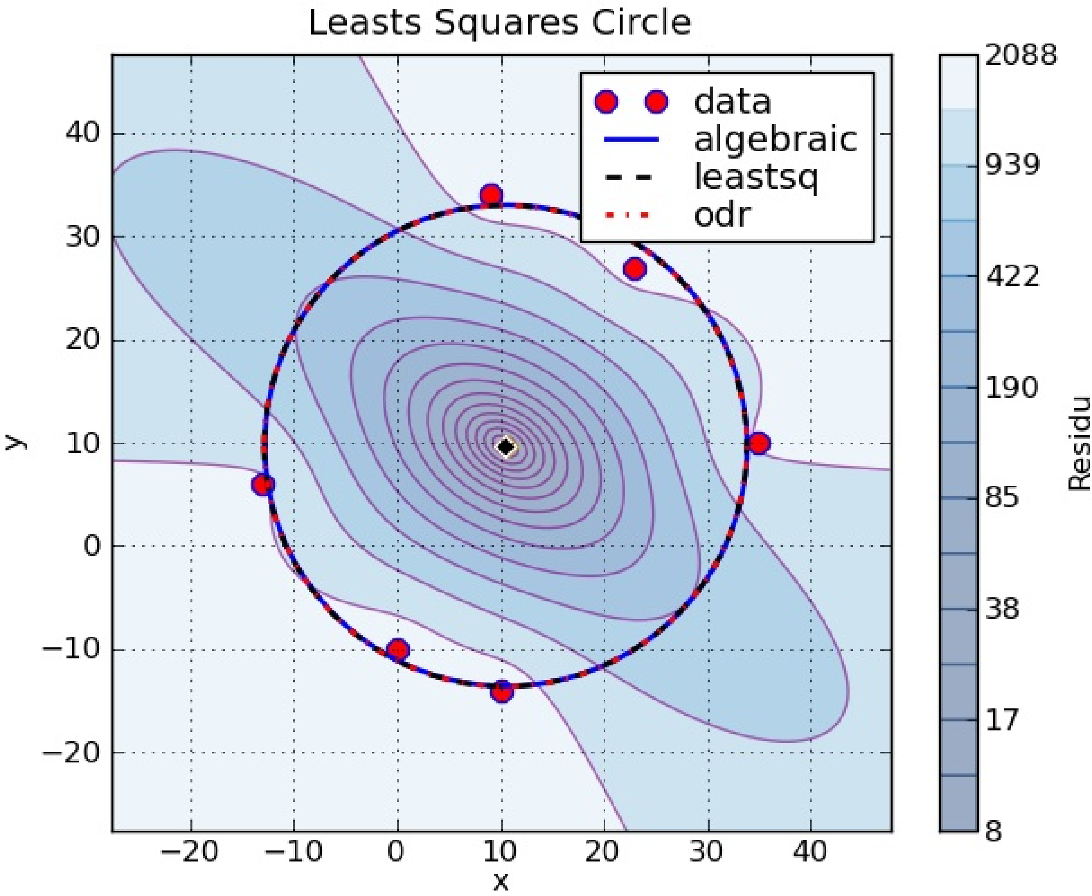


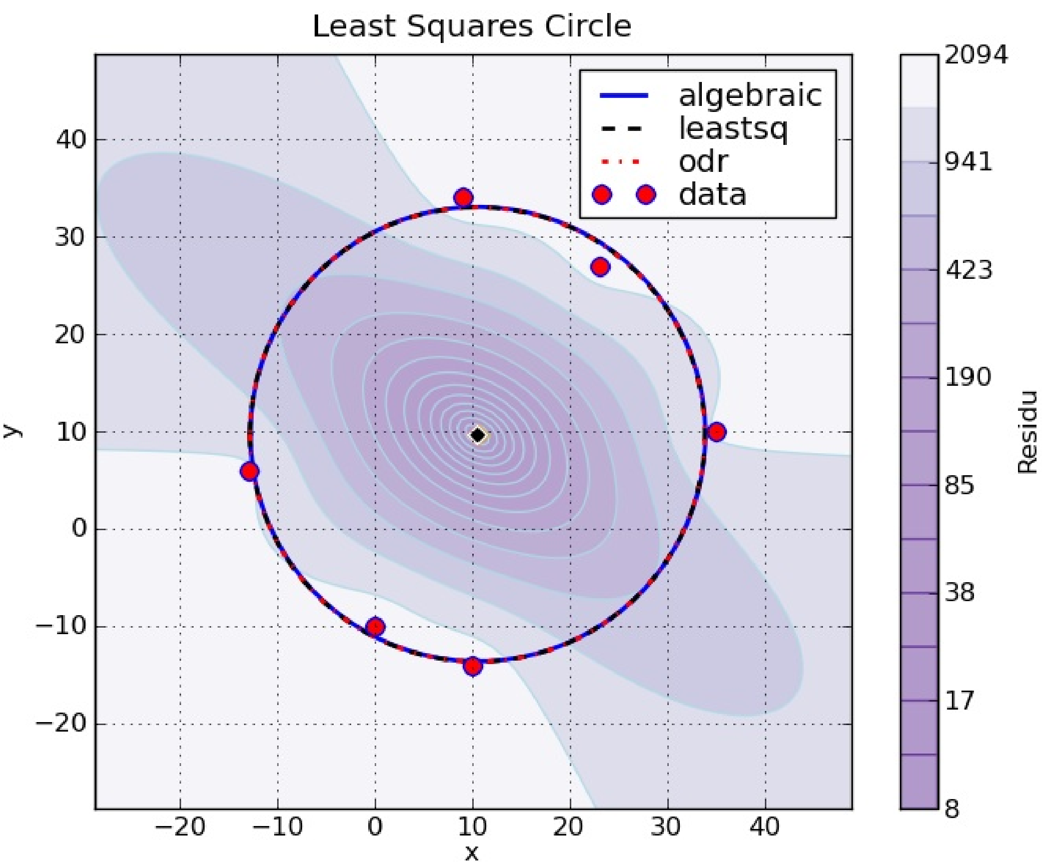
Least Squares Circle
algebraic
leastsq
odr
data
x
y
Residu_2 - algebraic approximation
26400735
9203246
3208234
1118384
389867
135907
47377
16515


Leasts Squares Circle
data
algebraic
leastsq
odr
2093.6
941.1
423.0
190.1
85.5
38.4
17.3
7.8
Residu
y
x


Leasts Squares Circle
data
algebraic
leastsq
odr
2108
947
425
191
86
39
17
8
Residu
y
x


Leasts Squares Circle
data
algebraic
leastsq
odr
y
x
Residu
2088
939
422
190
85
38
17
8
40
30
20
10
0
−10
−20
−20
−10
0
10
20
30
40


Leasts Squares Circle
data
algebraic
leastsq
odr
y
x
Residu
2088
939
422
190
85
38
17
8
40
30
20
10
0
−10
−20
−20
−10
0
10
20
30
40


Least Squares Circle
algebraic
leastsq
odr
data
y
x
Residu
2094
941
423
190
85
38
17
8
40
30
20
10
0
−10
−20
−20
−10
0
10
20
30
40

Linear regression

Linear Regression Example

```
from scipy import linspace, polyval, polyfit, sqrt, stats, randn
from pylab import plot, title, show , legend

#Linear regression example
# This is a very simple example of using two scipy tools
# for linear regression, polyfit and stats.linregress

#Sample data creation
#number of points
n=50
t=linspace(-5,5,n)
#parameters
a=0.8; b=-4
x=polyval([a,b],t)
#add some noise
xn=x+randn(n)

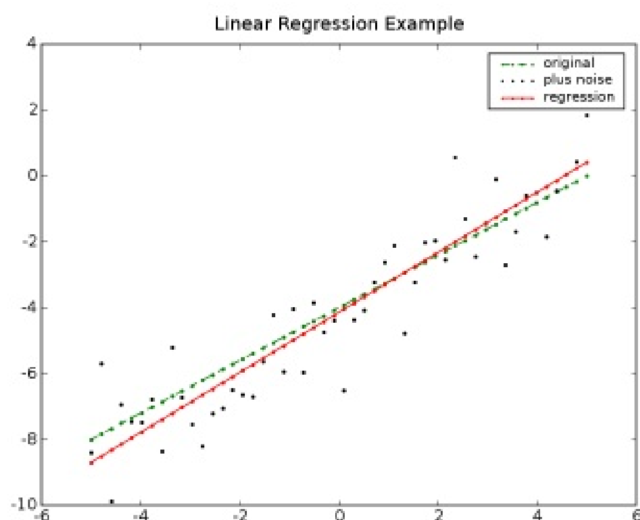
#Linear regressison -polyfit - polyfit can be used other orders pol
(ar,br)=polyfit(t,xn,1)
xr=polyval([ar,br],t)
#compute the mean square error
err=sqrt(sum((xr-xn)**2)/n)

print('Linear regression using polyfit')
print('parameters: a=%.2f b=%.2f  \nregression: a=%.2f b=%.2f, ms e

#matplotlib ploting
title('Linear Regression Example')
plot(t,x,'g.--')
plot(t,xn,'k.')
plot(t,xr,'r.-')
legend(['original','plus noise', 'regression'])

show()

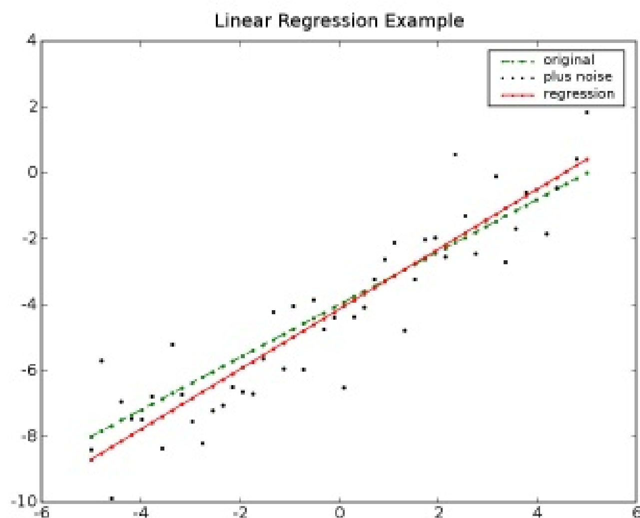
#Linear regression using stats.linregress
(a_s,b_s,r,tt,stderr)=stats.linregress(t,xn)
print('Linear regression using stats.linregress')
print('parameters: a=%.2f b=%.2f  \nregression: a=%.2f b=%.2f, std
```



Another example:
[http://www2.warwick.ac.uk/fac/sci/moac/currentstudents/peter_cock/python/lin_reg using scipy (and R) to calculate Linear Regressions]

Attachments

- `linregress.png`



OLS

OLS is an abbreviation for ordinary least squares.

The class estimates a multi-variate regression model and provides a variety of fit-statistics. To see the class in action download the `ols.py` file and run it (python ols.py). This)# will estimate a multi-variate regression using simulated data and provide output. It will also provide output from R to validate the results if you have rpy installed (http://rpy.sourceforge.net/).

To import the class:

```
#!python
import ols
```

After importing the class you can estimate a model by passing data to it as follows

```
#!python
mymodel = ols.ols(y,x,y_varnm,x_varnm)
```

where y is an array with data for the dependent variable, x contains the independent variables, y_varnm, is a string with the variable label for the dependent variable, and x_varnm is a list of variable labels for the independent variables. Note: An intercept term and variable label is automatically added to the model.

Example Usage

```
#!python
>>> import ols
>>> from numpy.random import randn
>>> data = randn(100,5)
>>> y = data[:,0]
>>> x = data[:,1:]
>>> mymodel = ols.ols(y,x,'y',['x1','x2','x3','x4'])
>>> mymodel.p               # return coefficient p-values
array([ 0.31883448,  0.7450663 ,  0.95372471,  0.97437927,  0.0999
>>> mymodel.summary()       # print results
===================================================================
Dependent Variable: y
Method: Least Squares
Date: Thu, 28 Feb 2008
Time: 22:32:24
# obs:               100
# variables:           5
===================================================================
variable     coefficient     std. Error      t-statistic     prob.
===================================================================
const           0.107348      0.107121      1.002113      0.318834
x1             -0.037116      0.113819     -0.326100      0.745066
x2              0.006657      0.114407      0.058183      0.953725
x3              0.003617      0.112318      0.032201      0.974379
x4              0.186022      0.111967      1.661396      0.099931
===================================================================
Models stats                         Residual stats
===================================================================
R-squared                  0.033047         Durbin-Watson stat   2.0129
Adjusted R-squared        -0.007667         Omnibus stat         5.6643
F-statistic                0.811684         Prob(Omnibus stat)   0.0588
Prob (F-statistic)         0.520770             JB stat              6.
Log likelihood          -145.182795         Prob(JB)             0.0471
AIC criterion              3.003656         Skew                 0.3271
BIC criterion              3.133914         Kurtosis             4.0189
===================================================================
```

== Note ==

Library function [http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html numpy.linalg.lstsq()] performs basic OLS estimation.

Attachments

- `ols.0.1.py`
- `ols.0.2.py`

Optimization and fit demo

This is a quick example of creating data from several Bessel functions and finding local maxima, then fitting a curve using some spline functions from the scipy.interpolate module. The enthought.chaco package and wxpython are used for creating the plot. PyCrust (which comes with wxpython) was used as the python shell.

```
from enthought.chaco.wx import plt
from scipy import arange, optimize, special

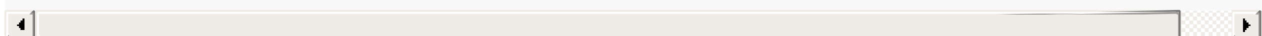
plt.figure()
plt.hold()
w = []
z = []
x = arange(0,10,.01)

for k in arange(1,5,.5):
   y = special.jv(k,x)
   plt.plot(x,y)
   f = lambda x: -special.jv(k,x)
   x_max = optimize.fminbound(f,0,6)
   w.append(x_max)
   z.append(special.jv(k,x_max))

plt.plot(w,z, 'ro')
from scipy import interpolate
t = interpolate.splrep(w, z, k=3)
s_fit3 = interpolate.splev(x,t)
plt.plot(x,s_fit3, 'g-')
t5 = interpolate.splrep(w, z, k=5)
s_fit5 = interpolate.splev(x,t5)
plt.plot(x,s_fit5, 'y-')
```

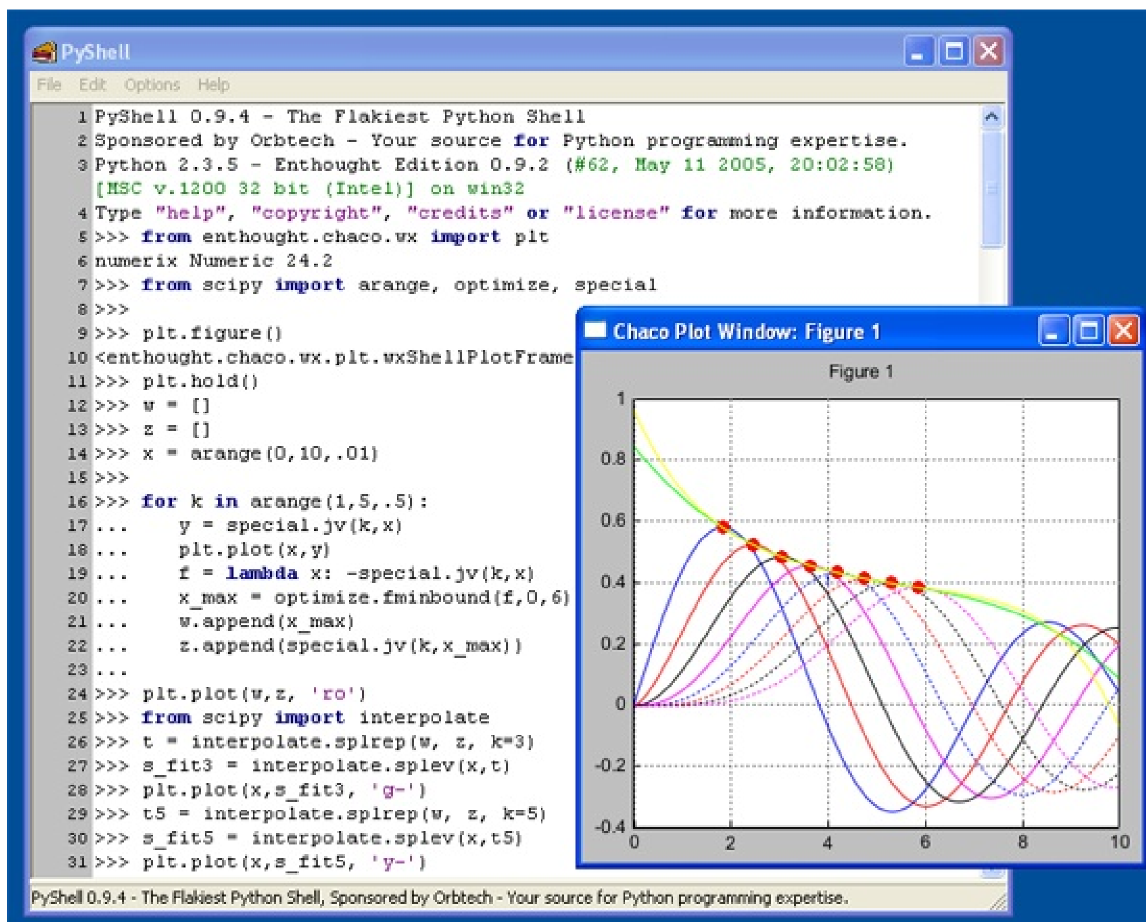
```
#class left

# ![](files/attachments/OptimizationAndFitDemo1/chacoscreenshot.png
Optimization Example
```



Attachments

- `chacoscreenshot.png`


PyShell
File Edit Options Help
1 PyShell 0.9.4 - The Flakiest Python Shell
2 Sponsored by Orbtech - Your source for Python programming expertise.
3 Python 2.3.5 - Enthought Edition 0.9.2 (#62, May 11 2005, 20:02:58)
[MSC v.1200 32 bit (Intel)] on win32
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>> from enthought.chaco.wx import plt
6 numerix Numeric 24.2
7 >>> from scipy import arange, optimize, special
8 >>>
9 >>> plt.figure()
10 <enthought.chaco.wx.plt.wxShellPlotFrame
11 >>> plt.hold()
12 >>> w = []
13 >>> z = []
14 >>> x = arange(0,10,.01)
15 >>>
16 >>> for k in arange(1,5,.5):
17 ...   y = special.jv(k,x)
18 ...   plt.plot(x,y)
19 ...   f = lambda x: -special.jv(k,x)
20 ...   x_max = optimize.fminbound(f,0,6)
21 ...   w.append(x_max)
22 ...   z.append(special.jv(k,x_max))
23 ...
24 >>> plt.plot(w,z, 'ro')
25 >>> from scipy import interpolate
26 >>> t = interpolate.splrep(w, z, k=3)
27 >>> s_fit3 = interpolate.splev(x,t)
28 >>> plt.plot(x,s_fit3, 'g-')
29 >>> t5 = interpolate.splrep(w, z, k=5)
30 >>> s_fit5 = interpolate.splev(x,t5)
31 >>> plt.plot(x,s_fit5, 'y-')
PyShell 0.9.4 - The Flakiest Python Shell, Sponsored by Orbtech - Your source for Python programming expertise.
Chaco Plot Window: Figure 1
Figure 1
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
0
2
4
6
8
10

Optimization demo

SciPy’s optimization package is scipy.optimize. The most basic non-linear optimization functions are:

```
* optimize.fmin(func, x0), which finds the minimum of func(x) start
* optimize.fsolve(func, x0), which finds a solution to func(x) = 0
* optimize.fminbound(func, x1, x2), which finds the minimum of a sc
```

See the scipy.optimze documentation for details.

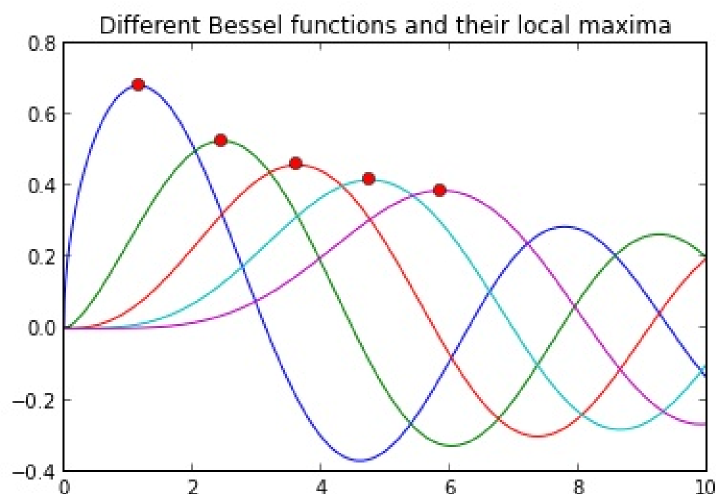
This is a quick demonstration of generating data from several Bessel functions and finding some local maxima using fminbound. This uses ipython with the -pylab switch.

```
from scipy import optimize, special
from numpy import *
from pylab import *

x = arange(0,10,0.01)

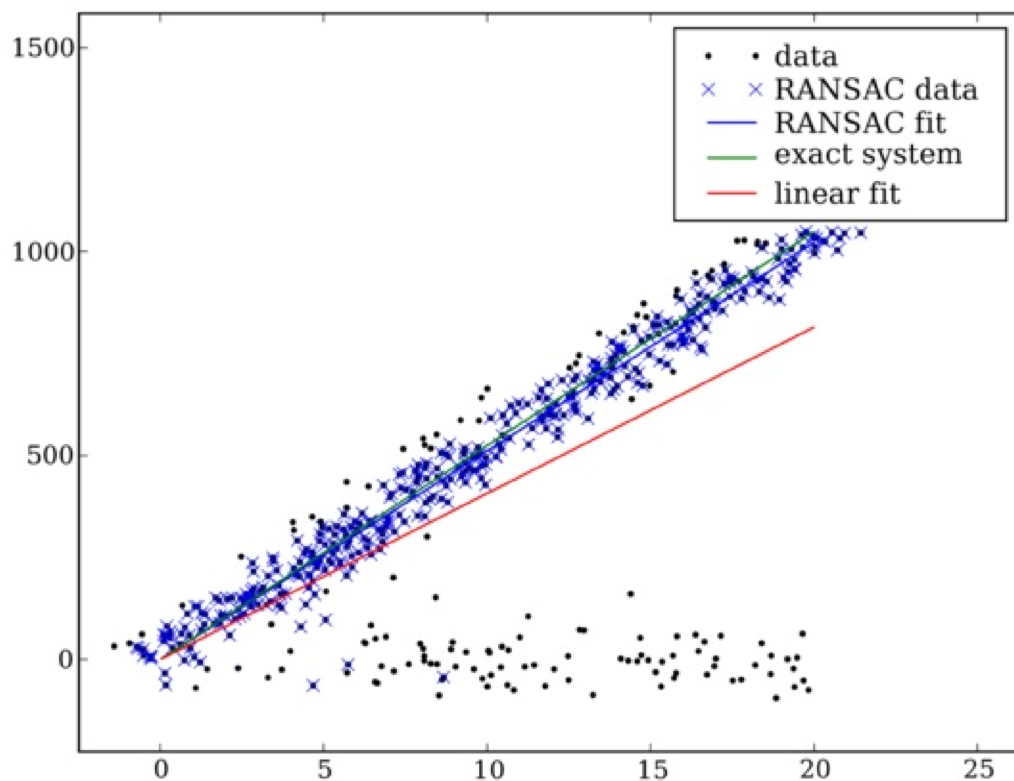
for k in arange(0.5,5.5):
     y = special.jv(k,x)
     plot(x,y)
     f = lambda x: -special.jv(k,x)
     x_max = optimize.fminbound(f,0,6)
     plot([x_max], [special.jv(k,x_max)],'ro')

title('Different Bessel functions and their local maxima')
show()
```



RANSAC

The attached file `ransac.py` implements the RANSAC algorithm. An example image:



To run the file, save it to your computer, start IPython

```
ipython -wthread
```

Import the module and run the test program

```
import ransac
ransac.test()
```

To use the module you need to create a model class with two methods

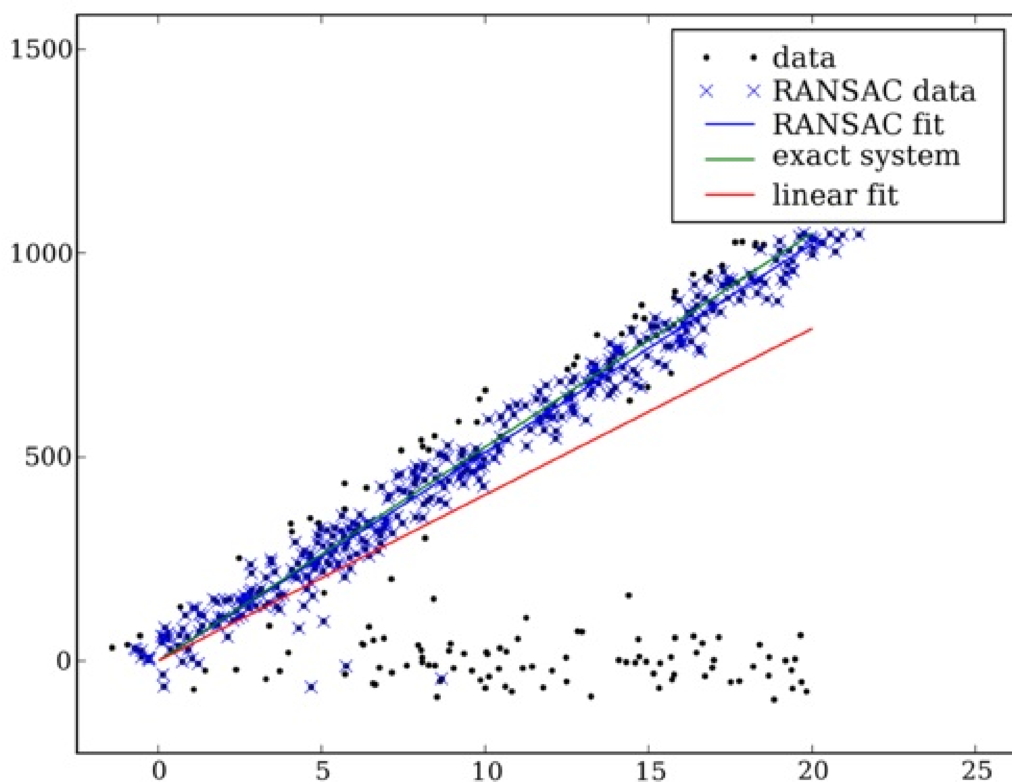
```
def fit(self, data):
  """Given the data fit the data with your model and return the moc
def get_error(self, data, model):
  """Given a set of data and a model, what is the error of using tl
```

An example of such model is the class LinearLeastSquaresModel as seen the file source (below)

`ransac.py`

Attachments

- `ransac.png`
- `ransac.py`



Robust nonlinear regression in scipy

One of the main applications of nonlinear least squares is nonlinear regression or curve fitting. That is by given pairs `\(\left\{ (t_i, y_i) \: i = 1, \ldots, n \right\}\)` estimate parameters `\(\mathbf{x}\)` defining a nonlinear function `\(\varphi(t; \mathbf{x})\)`, assuming the model:

```
\[\begin{equation} y_i = \varphi(t_i; \mathbf{x}) + \epsilon_i \end
```

Where `\(\epsilon_i\)` is the measurement (observation) errors. In the least-squares estimation we search `\(\mathbf{x}\)` as the solution of the following optimization problem:

```
\[\begin{equation} \frac{1}{2} \sum_{i=1}^n (\varphi(t_i; \mathbf{x
```

Such formulation is intuitive and convinient from mathematical point of view. From the probabilistic point of view the least-squares solution is known to be the maximum likelihood estimate, provided that all `\(\epsilon_i\)` are independent and normally distributed random variables.

So theoretically it is not optimal when `\(\epsilon_i\)` have distribution other than normal. Although in engineering practice it is usally not important, i.e. if errors behave as some reasonable random variables with zero mean a result of least-squares estimation will be satisfactory.

The real problems start when data is contaminated by outliers (completly wrong measurements). In this case the least-squares solution can become significantly biased to avoid very high residuals on outliers. To qualitatively explain why it is happening, let's consider the simplest least-squares problem:

```
\[\begin{align} \frac{1}{2} \sum_{i=1}^n (a - x_i)^2 \rightarrow \r
```

And the solution is the mean value:

```
\[\begin{align} a^* = \frac{1}{n} \sum_{i=1}^n x_i \end{align}\]
```

Now imagine:
`\(a = 1, n=4, x_1 = 0.9, x_2 = 1.05, x_3=0.95, x_4=10 \text{ (outli`
. The solution $a^* = 3.225$, i.e. completely ruined by a single outlier.

One of the well known robust estimators is l1-estimator, in which the sum of absolute values of the residuals is minimized. For demonstration, again consider the simplest problem:

```
\[\begin{equation} \sum_{i=1}^n |a - x_i| \rightarrow \min_a \end{e
```

And the well-known solution is the median of $\{x_i\}$, such that the small number of outliers won't affect the solution. In our toy problem we have $a^* = 1$.

The only disadvantage of l1-estimator is that arising optimization problem is hard, as the function is nondifferentiable everywhere, which is particularly troublesome for efficient nonlinear optimization. It means that we are better to stay with differentiable problems, but somehow incorporate robustness in estimation. To accomplish this we introduce a sublinear function $\rho(z)$ (i.e. its growth should be slower than linear) and formulate a new least-squares-like optimization problem:

```
\[\begin{equation} \frac{1}{2} \sum_{i=1}^n \rho \left((\varphi(t_i
```

Turns out that this problem can be reduced to standard nonlinear least squares by modifying a vector of residuals and Jacobian matrix on each iteration, such that computed gradient and Hessian appxoximation match the ones of the objective function. Refer to the paper for details.

The implemented choices of ρ are the following:

1. Linear functior which gives a standard least squares: $\rho(z) = z$.
2. Huber loss:
   `\(\rho(z) = \begin{cases} z & z \leq 1 \\ \sqrt{z} - 1 & z >`
3. Smooth approximation to absolute value loss, "soft l1 loss":
   $\rho(z) = 2 (\sqrt{1 + z} - 1)$
4. Cauchy loss: $\rho(z) = \ln(1 + z)$.
5. Loss by arctan: $\rho(z) = \arctan z$.

The functions 2 and 3 are relatively mild and give approximately absolute value loss for large residuals. The last two functions are strongly sublinear and give significant attenuation for outliers.

The loss functions above are written with the assumption that the soft threshold between inliners and outliers is equal to 1.0. To generalize, we introduce the scaling parameter `\(C\)` and evaluate the loss as

```
\[\begin{equation} \hat{\rho}(r^2) = C^2 \rho \left( \left(\frac{r
```

Note that if residuals are small, we have `\(\hat{\rho}(r^2) \approx \rho(r^2) \approx r^2\)` for any `\(\rho(z)\)` defined above.

To illustrate the behaviour of the introduced functions we build a plot:

```
import numpy as np

%matplotlib inline
import matplotlib.pyplot as plt

from matplotlib import rcParams
rcParams['figure.figsize'] = (10, 6)
rcParams['legend.fontsize'] = 16
rcParams['axes.labelsize'] = 16
```

```
r = np.linspace(0, 5, 100)

linear = r**2

huber = r**2
huber[huber > 1] = 2 * r[huber > 1] - 1

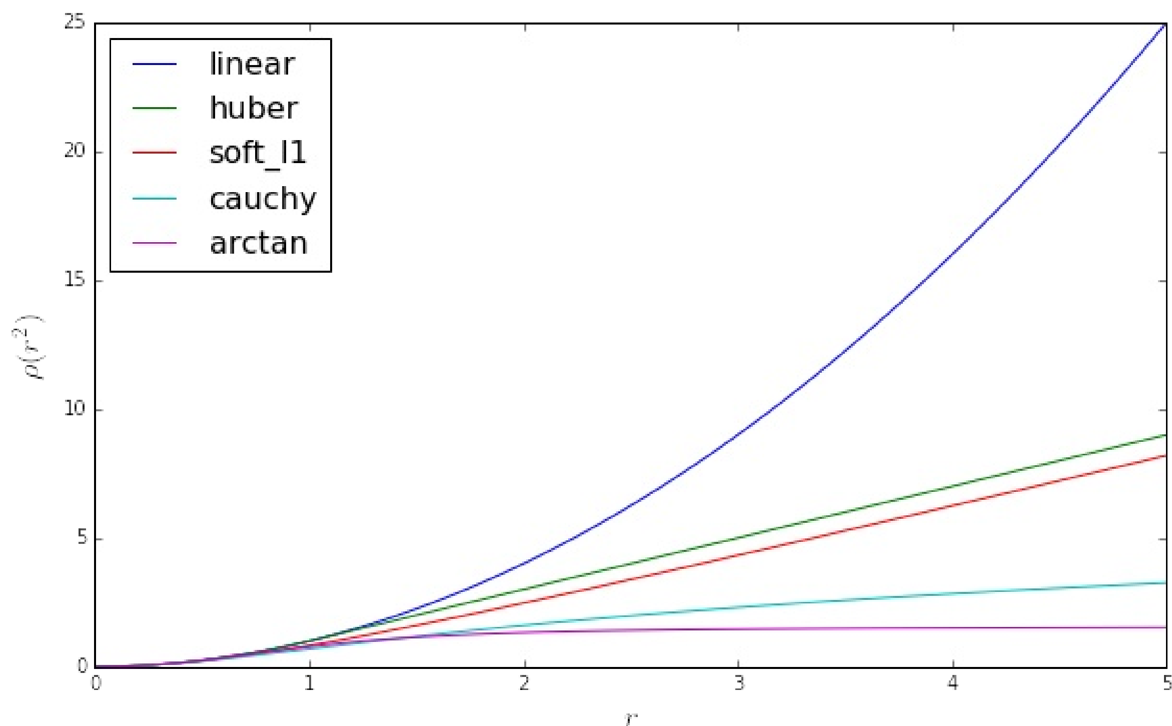
soft_l1 = 2 * (np.sqrt(1 + r**2) - 1)

cauchy = np.log1p(r**2)

arctan = np.arctan(r**2)
```

```
plt.plot(r, linear, label='linear')
plt.plot(r, huber, label='huber')
plt.plot(r, soft_l1, label='soft_l1')
plt.plot(r, cauchy, label='cauchy')
plt.plot(r, arctan, label='arctan')
plt.xlabel("$r$")
plt.ylabel(r"$\rho(r^2)$")
plt.legend(loc='upper left')
```

```
<matplotlib.legend.Legend at 0x1058c1ef0>
```



Now we will show how robust loss functions work on a model example. We define the model function as

```
\[\begin{equation} f(t; A, \sigma, \omega) = A e^{-\sigma t} \sin (
```

Which can model a observed displacement of a linear damped oscillator.

Define data generator:

```
def generate_data(t, A, sigma, omega, noise=0, n_outliers=0, randor
    y = A * np.exp(-sigma * t) * np.sin(omega * t)
    rnd = np.random.RandomState(random_state)
    error = noise * rnd.randn(t.size)
    outliers = rnd.randint(0, t.size, n_outliers)
    error[outliers] *= 35
    return y + error
```

Define model parameters:

```
A = 2
sigma = 0.1
omega = 0.1 * 2 * np.pi
x_true = np.array([A, sigma, omega])

noise = 0.1

t_min = 0
t_max = 30
```

Data for fitting the parameters will contain 3 outliers:

```
t_train = np.linspace(t_min, t_max, 30)
y_train = generate_data(t_train, A, sigma, omega, noise=noise, n_ou
```

Define the function computing residuals for least-squares minimization:

```
def fun(x, t, y):
    return x[0] * np.exp(-x[1] * t) * np.sin(x[2] * t) - y
```

Use all ones as the initial estimate.

```
x0 = np.ones(3)
```

```
from scipy.optimize import least_squares
```

Run standard least squares:

```
res_lsq = least_squares(fun, x0, args=(t_train, y_train))
```

Run robust least squares with `loss='soft_l1'`, set `f_scale` to 0.1 which means that inlier residuals are approximately lower than 0.1.

```
res_robust = least_squares(fun, x0, loss='soft_l1', f_scale=0.1, ar
```

Define data to plot full curves.

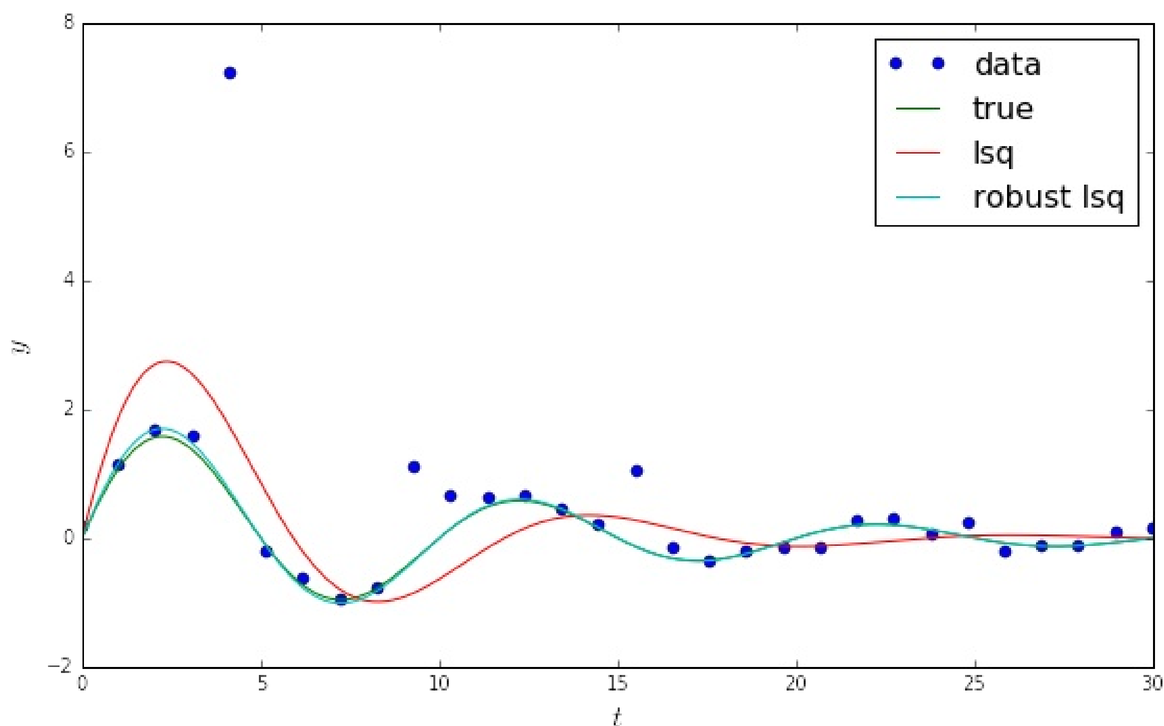
```
t_test = np.linspace(t_min, t_max, 300)
y_test = generate_data(t_test, A, sigma, omega)
```

Compute predictions with found parameters:

```
y_lsq = generate_data(t_test, *res_lsq.x)
y_robust = generate_data(t_test, *res_robust.x)
```

```
plt.plot(t_train, y_train, 'o', label='data')
plt.plot(t_test, y_test, label='true')
plt.plot(t_test, y_lsq, label='lsq')
plt.plot(t_test, y_robust, label='robust lsq')
plt.xlabel('$t$')
plt.ylabel('$y$')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x107031898>
```



We clearly see that standard least squares react significantly on outliers and give very biased solution, whereas robust least squares (with soft l1-loss) give the solution very close to the true model.

Solving a discrete boundary-value problem in scipy

Mathematical formulation

We consider a nonlinear elliptic boundary-value problem in a square domain $\Omega = [0, 1] \times [0, 1]$:

```
\[\begin{split}\Delta u + k f(u) = 0 \\ u = 0 \text{ on } \partial
```

Here $u = u(x,y)$ is an unknown function, Δ is Laplace operator, k is some constant and $f(u)$ is a given function.

A usual computational approach to such problems is discretization. We use a uniform grid with some step h , and define $u_{i, j} = u(i h , jh)$. By approximating Laplace operator using 5-point finite difference we get a system of equations in the form:

```
\[\begin{split}u_{i - 1, j} + u_{i + 1, j} + u_{i, j - 1} + u_{i, j
```

Here $c = k h^2$.

Defining the problem for scipy

From now on we focus on the discrete version and consider a grid with $n = 100$ ticks for each dimension and set $c = 1$ and $f(u) = u^3$.

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import least_squares
from scipy.sparse import coo_matrix
```

```
n = 100
c = 1
```

```
def f(u):
    return u**3

def f_prime(u):
    return 3 * u**2
```

To solve the system of equations we will use `scipy.optimize.least_squares`.

We define a function computing left-hand sides of each equation. Note that we assume values on the boundary to be fixed at zeros and don't change them during optimization.

```
def fun(u, n, f, f_prime, c, **kwargs):
    v = np.zeros((n + 2, n + 2))
    u = u.reshape((n, n))
    v[1:-1, 1:-1] = u
    y = v[:-2, 1:-1] + v[2:, 1:-1] + v[1:-1, :-2] + v[1:-1, 2:] - 4
    return y.ravel()
```

It is always recommened to provide analytical Jacobian if possible. In our problem we have `\(n^2=10000\)` equations and variables, but each equation depends only on few variables, thus we should compute Jacobian in a sparse format.

It is convenient to precompute indices of rows and columns of nonzero elements in Jacobian. We definte the corresponding function:

```
def compute_jac_indices(n):
    i = np.arange(n)
    jj, ii = np.meshgrid(i, i)

    ii = ii.ravel()
    jj = jj.ravel()

    ij = np.arange(n**2)

    jac_rows = [ij]
    jac_cols = [ij]

    mask = ii > 0
    ij_mask = ij[mask]
    jac_rows.append(ij_mask)
    jac_cols.append(ij_mask - n)

    mask = ii < n - 1
    ij_mask = ij[mask]
    jac_rows.append(ij_mask)
    jac_cols.append(ij_mask + n)

    mask = jj > 0
    ij_mask = ij[mask]
    jac_rows.append(ij_mask)
    jac_cols.append(ij_mask - 1)

    mask = jj < n - 1
    ij_mask = ij[mask]
    jac_rows.append(ij_mask)
    jac_cols.append(ij_mask + 1)

    return np.hstack(jac_rows), np.hstack(jac_cols)
```

After that computing Jacobian in `coo_matrix` format is simple:

```
jac_rows, jac_cols = compute_jac_indices(n)
```

```
def jac(u, n, f, f_prime, c, jac_rows=None, jac_cols=None):
    jac_values = np.ones_like(jac_cols, dtype=float)
    jac_values[:n**2] = -4 + c * f_prime(u)
    return coo_matrix((jac_values, (jac_rows, jac_cols)), shape=(n
```

Solving the problem

Without any insight to the problem we set all the values to 0.5 initially. Note, that it is not guranteed that the given continuous or discrete problems have a unique solution.

```
u0 = np.ones(n**2) * 0.5
```

Precompute rows and columns of nonzero elements in Jacobian:

```
jac_rows, jac_cols = compute_jac_indices(n)
```

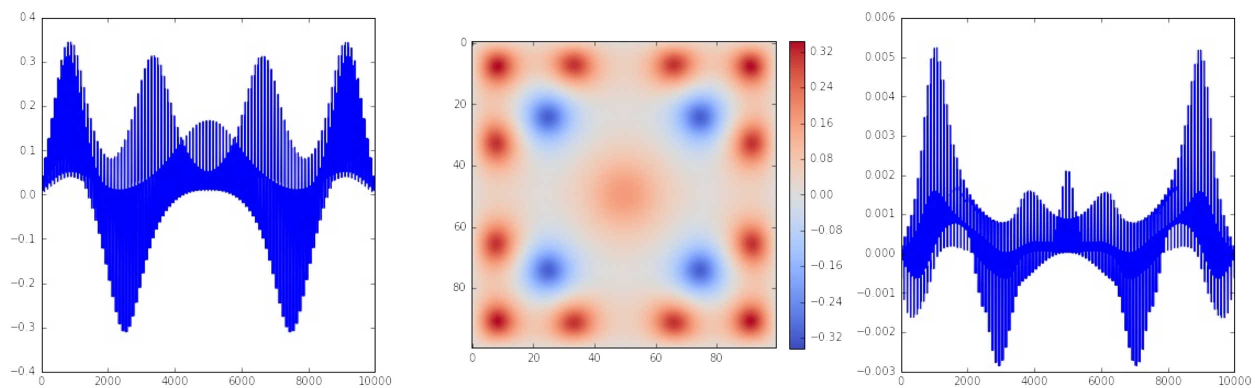
Now we are ready to run the optimization. The first solution will be computed without imposing any bounds on the variables.

```
res_1 = least_squares(fun, u0, jac=jac, gtol=1e-3, args=(n, f, f_pr
```

```
 gtol termination condition is satisfied.
 Function evaluations 106, initial cost 1.0412e+02, final cost 5.27
```

Below we visualize the first solution. The left plot shows the flatten solution, the middle plot shows how the solution looks in the square domain, and the right plot shows final residuals in each node.

```
plt.figure(figsize=(16, 5))
plt.subplot(132)
plt.imshow(res_1.x.reshape((n, n)), cmap='coolwarm', vmin=-max(abs(
plt.colorbar(use_gridspec=True, fraction=0.046, pad=0.04)
plt.subplot(131)
plt.plot(res_1.x)
plt.subplot(133)
plt.plot(res_1.fun)
plt.tight_layout()
```

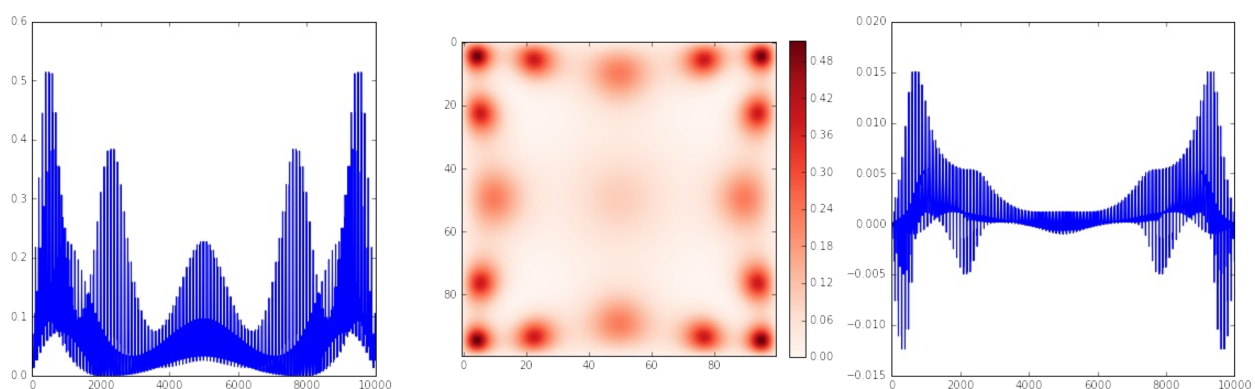


It is possible that some physical considerations require that the solution must be non-negative everywhere. We can achieve that by specifying bounds to the solver:

```
res_2 = least_squares(fun, u0, jac=jac, bounds=(0, np.inf), gtol=1e
                      args=(n, f, f_prime, c), kwargs={'jac_rows':
```

```
 gtol termination condition is satisfied.
 Function evaluations 34, initial cost 1.0412e+02, final cost 4.134
```

```
plt.figure(figsize=(16, 5))
plt.subplot(132)
plt.imshow(res_2.x.reshape((n, n)), cmap='Reds')
plt.colorbar(use_gridspec=True, fraction=0.046, pad=0.04)
plt.subplot(131)
plt.plot(res_2.x)
plt.subplot(133)
plt.plot(res_2.fun)
plt.tight_layout()
```



We see that setting a lower bound allowed us to find a different non-negative solution.

You can try running optimization from different starting points, using different bounds or changing `\(c\)` and `\(f(u)\)`, and see how it affects the result from `least_squares` solver.

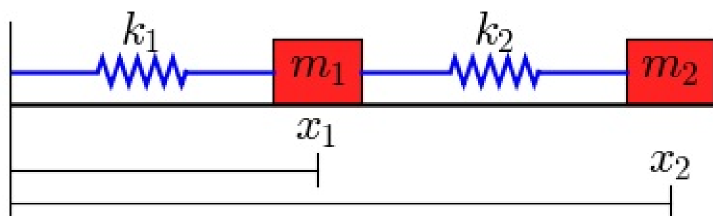
Numpy & Scipy / Ordinary differential equations

Coupled spring-mass system

This cookbook example shows how to solve a system of differential equations. (Other examples include the *Lotka-Volterra Tutorial*, the *Zombie Apocalypse* and the `KdV example`.)

A Coupled Spring-Mass System

This figure shows the system to be modeled:



Two objects with masses `\(m_1\)` and `\(m_2\)` are coupled through springs with spring constants `\(k_1\)` and `\(k_2\)`. The left end of the left spring is fixed. We assume that the lengths of the springs, when subjected to no external forces, are `\(L_1\)` and `\(L_2\)`.

The masses are sliding on a surface that creates friction, so there are two friction coefficients, b,,1,, and b,,2,,.

The differential equations for this system are

```
\(m_1 x_1'' + b_1 x_1' + k_1 (x_1 - L_1) - k_2 (x_2 - x_1 - L_2) = (
```

```
\(m_2 x_2'' + b_2 x_2' + k_2 (x_2 - x_1 - L_2) = 0\)
```

This is a pair of coupled second order equations. To solve this system with one of the ODE solvers provided by SciPy, we must first convert this to a system of first order differential equations. We introduce two variables

`\(y_1 = x_1'\)`, `\(y_2 = x_2'\)`

These are the velocities of the masses.

With a little algebra, we can rewrite the two second order equations as the following system of four first order equations:

```
\(x_1' = y_1\)
```

```
\(y_1' = (-b_1 y_1 - k_1 (x_1 - L_1) + k_2 (x_2 - x_1 - L_2))/m_1\)
```

```
\(x_2' = y_2\)
```

`\(y_2' = (-b_2 y_2 - k_2 (x_2 - x_1 - L_2))/m_2\)`

These equations are now in a form that we can implement in Python.

The following code defines the “right hand side” of the system of equations (also known as a vector field). I have chosen to put the function that defines the vector field in its own module (i.e. in its own file), but this is not necessary. Note that the arguments of the function are configured to be used with the function: the time t is the second argument.

```
def vectorfield(w, t, p):
    """
 Defines the differential equations for the coupled spring-mass sys

 Arguments:
 w :  vector of the state variables:
 w = [x1,y1,x2,y2]
 t :  time
 p :  vector of the parameters:
 p = [m1,m2,k1,k2,L1,L2,b1,b2]
 """
    x1, y1, x2, y2 = w
    m1, m2, k1, k2, L1, L2, b1, b2 = p

    # Create f = (x1',y1',x2',y2'):
    f = [y1,
         (-b1 * y1 - k1 * (x1 - L1) + k2 * (x2 - x1 - L2)) / m1,
         y2,
         (-b2 * y2 - k2 * (x2 - x1 - L2)) / m2]
    return f
```

Next, here is a script that uses to solve the equations for a given set of parameter values, initial conditions, and time interval. The script prints the points in the solution to the terminal. Normally you will redirect its output to a file.

```
# Use ODEINT to solve the differential equations defined by the vec
from scipy.integrate import odeint

# Parameter values
# Masses:
m1 = 1.0
m2 = 1.5
# Spring constants
k1 = 8.0
k2 = 40.0
# Natural lengths
L1 = 0.5
L2 = 1.0
# Friction coefficients
b1 = 0.8
b2 = 0.5

# Initial conditions
# x1 and x2 are the initial displacements; y1 and y2 are the initia
x1 = 0.5
y1 = 0.0
x2 = 2.25
y2 = 0.0

# ODE solver parameters
abserr = 1.0e-8
relerr = 1.0e-6
stoptime = 10.0
numpoints = 250

# Create the time samples for the output of the ODE solver.
# I use a large number of points, only because I want to make
# a plot of the solution that looks nice.
t = [stoptime * float(i) / (numpoints - 1) for i in range(numpoints

# Pack up the parameters and initial conditions:
p = [m1, m2, k1, k2, L1, L2, b1, b2]
w0 = [x1, y1, x2, y2]

# Call the ODE solver.
wsol = odeint(vectorfield, w0, t, args=(p,),
              atol=abserr, rtol=relerr)

with open('two_springs.dat', 'w') as f:
    # Print & save the solution.
    for t1, w1 in zip(t, wsol):
        print >> f, t1, w1[0], w1[1], w1[2], w1[3]
```

The following script uses Matplotlib to plot the solution generated by

```
# Plot the solution that was generated

from numpy import loadtxt
from pylab import figure, plot, xlabel, grid, hold, legend, title,
from matplotlib.font_manager import FontProperties

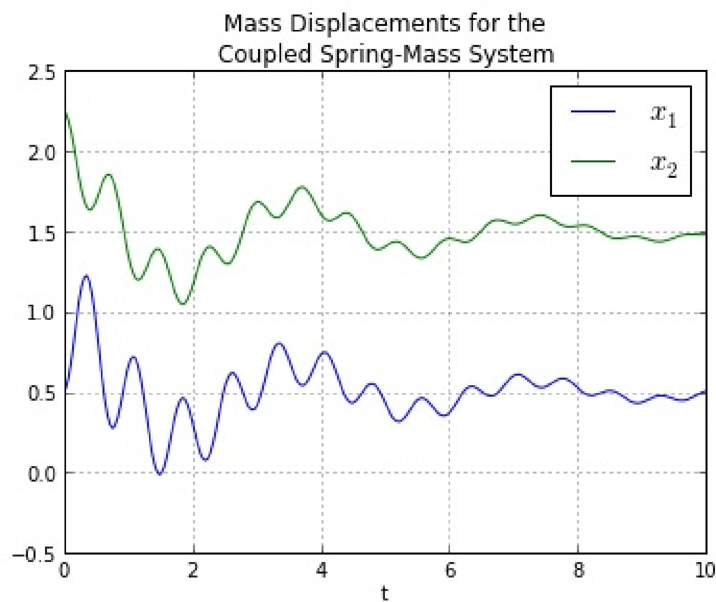
t, x1, xy, x2, y2 = loadtxt('two_springs.dat', unpack=True)

figure(1, figsize=(6, 4.5))

xlabel('t')
grid(True)
hold(True)
lw = 1

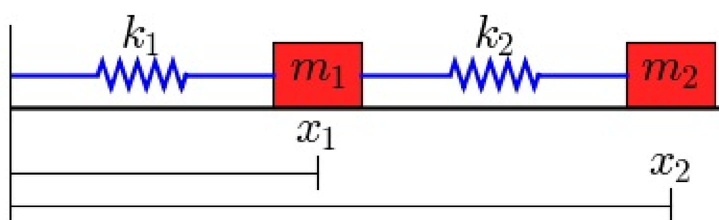
plot(t, x1, 'b', linewidth=lw)
plot(t, x2, 'g', linewidth=lw)

legend((r'$x_1$', r'$x_2$'), prop=FontProperties(size=16))
title('Mass Displacements for the\nCoupled Spring-Mass System')
savefig('two_springs.png', dpi=100)
```



Attachments

- `two_springs_diagram.png`


k_1
m_1
k_2
m_2
x_1
x_2

Korteweg de Vries equation

This page shows how the Korteweg-de Vries equation can be solved on a periodic domain using the method of lines, with the spatial derivatives computed using the pseudo-spectral method. In this method, the derivatives are computed in the frequency domain by first applying the FFT to the data, then multiplying by the appropriate values and converting back to the spatial domain with the inverse FFT. This method of differentiation is implemented by the **diff** function in the module **scipy.fftpack**.

We discretize the spatial domain, and compute the spatial derivatives using the **diff** function defined in the **scipy.fftpack** module. In the following code, this function is given the alias **psdiff** to avoid confusing it with the numpy function **diff**. By discretizing only the spatial dimension, we obtain a system of ordinary differential equations, which is implemented in the function **kdv(u, t, L)**. The function **kdv_solution(u0, t, L)** uses **scipy.integrate.odeint** to solve this system.

```
#!python

import numpy as np
from scipy.integrate import odeint
from scipy.fftpack import diff as psdiff

def kdv_exact(x, c):
    """Profile of the exact solution to the KdV for a single solito
    u = 0.5*c*np.cosh(0.5*np.sqrt(c)*x)**(-2)
    return u

def kdv(u, t, L):
    """Differential equations for the KdV equation, discretized in
    # Compute the x derivatives using the pseudo-spectral method.
    ux = psdiff(u, period=L)
    uxxx = psdiff(u, period=L, order=3)

    # Compute du/dt.
    dudt = -6*u*ux - uxxx

    return dudt

def kdv_solution(u0, t, L):
    """Use odeint to solve the KdV equation on a periodic domain.

  `u0` is initial condition, `t` is the array of time values at whic
  the solution is to be computed, and `L` is the length of the perio
  domain."""

    sol = odeint(kdv, u0, t, args=(L,), mxstep=5000)
    return sol
```

```
if __name__ == "__main__":
    # Set the size of the domain, and create the discretized grid.
    L = 50.0
    N = 64
    dx = L / (N - 1.0)
    x = np.linspace(0, (1-1.0/N)*L, N)

    # Set the initial conditions.
    # Not exact for two solitons on a periodic domain, but close er
    u0 = kdv_exact(x-0.33*L, 0.75) + kdv_exact(x-0.65*L, 0.4)

    # Set the time sample grid.
    T = 200
    t = np.linspace(0, T, 501)

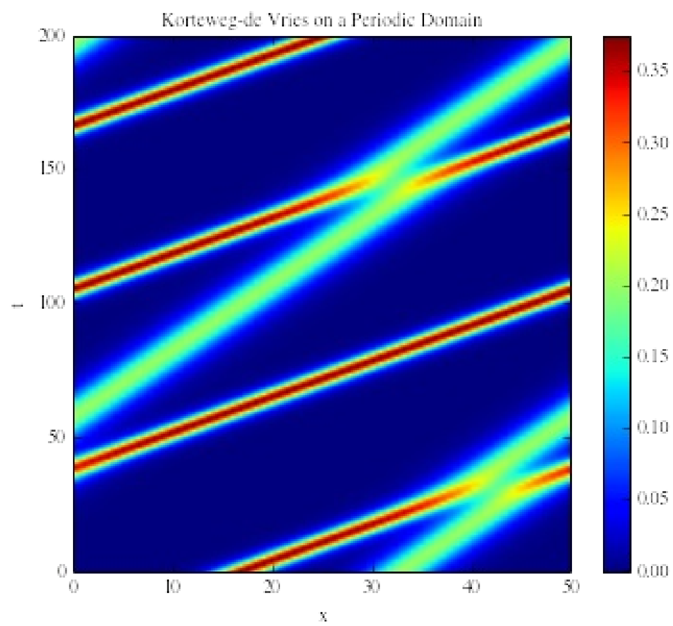
    print "Computing the solution."
    sol = kdv_solution(u0, t, L)

    print "Plotting."

    import matplotlib.pyplot as plt

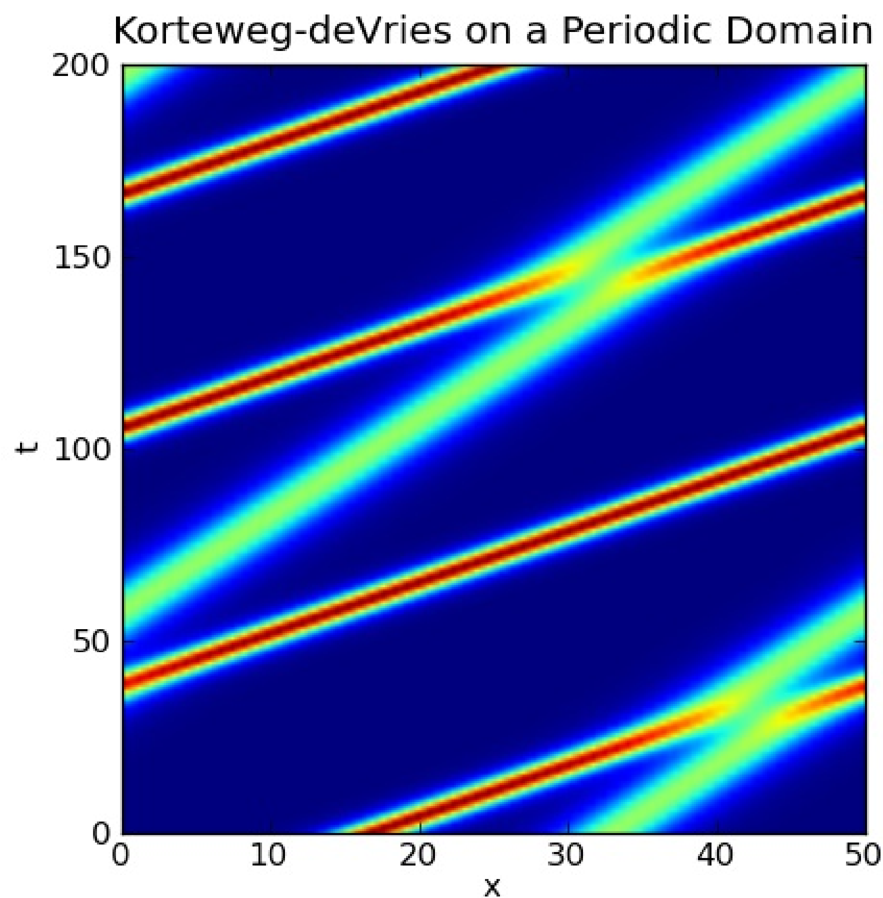
    plt.figure(figsize=(6,5))
    plt.imshow(sol[::-1, :], extent=[0,L,0,T])
    plt.colorbar()
    plt.xlabel('x')
    plt.ylabel('t')
    plt.axis('normal')
    plt.title('Korteweg-de Vries on a Periodic Domain')
    plt.show()
```

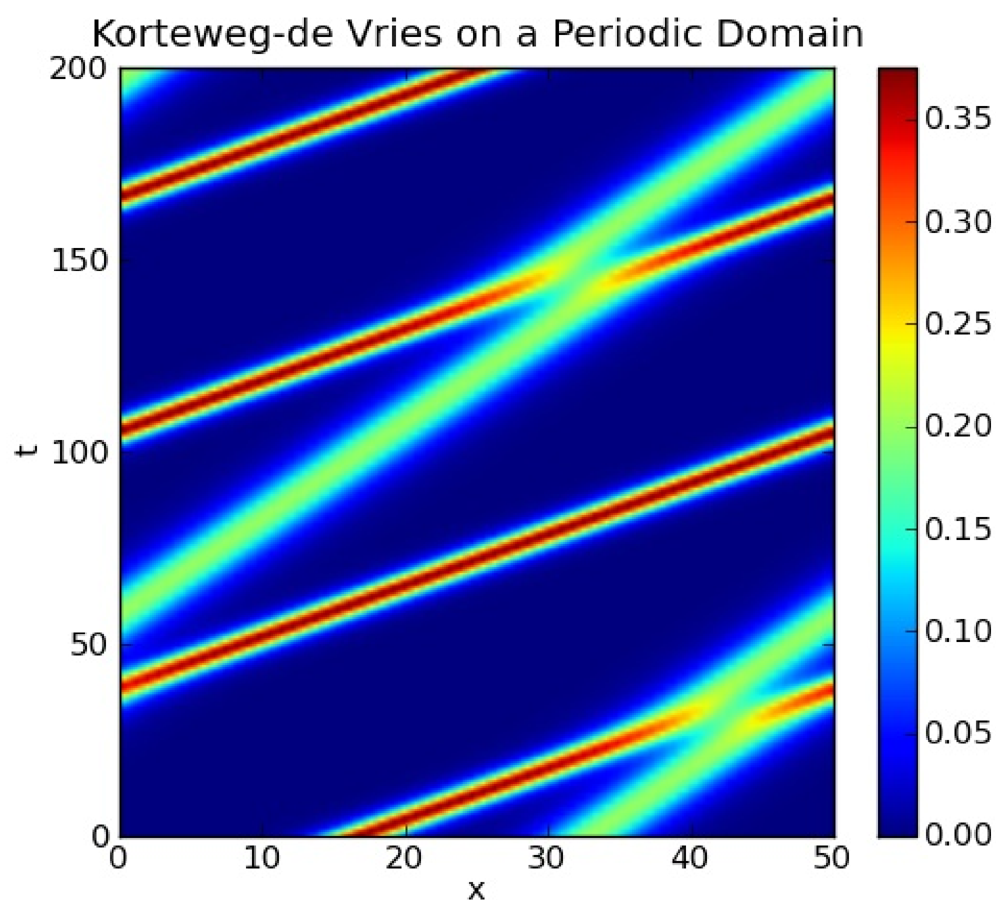
```
Computing the solution.
Plotting.
```



Attachments

- `kdv.png`
- `kdv2.png`


Korteweg-deVries on a Periodic Domain
200
150
t 100
50
0
0
10
20
30
40
50
x


Korteweg-de Vries on a Periodic Domain
200
150
t 100
50
0
0
10
20
30
40
50
x
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00

Matplotlib: lotka volterra tutorial

1.
   i. page was renamed from LoktaVolterraTutorial

This example describes how to integrate ODEs with the scipy.integrate module, and how to use the matplotlib module to plot trajectories, direction fields and other information.

You can get the source code for this tutorial here: `tutorial_lokta-voltera_v4.py` .

Presentation of the Lotka-Volterra Model

We will have a look at the Lotka-Volterra model, also known as the predator-prey equations, which is a pair of first order, non-linear, differential equations frequently used to describe the dynamics of biological systems in which two species interact, one a predator and the other its prey. The model was proposed independently by Alfred J. Lotka in 1925 and Vito Volterra in 1926, and can be described by

```
du/dt =  a*u -   b*u*v
dv/dt = -c*v + d*b*u*v
```

```
with the following notations:

*  u: number of preys (for example, rabbits)

*  v: number of predators (for example, foxes)

* a, b, c, d are constant parameters defining the behavior of the p

  + a is the natural growing rate of rabbits, when there's no fox

  + b is the natural dying rate of rabbits, due to predation

  + c is the natural dying rate of fox, when there's no rabbit

  + d is the factor describing how many caught rabbits let create a
```

We will use X=[u, v] to describe the state of both populations.

Definition of the equations:

```
#!python
from numpy import *
import pylab as p
# Definition of parameters
a = 1.
b = 0.1
c = 1.5
d = 0.75
def dX_dt(X, t=0):
    """ Return the growth rate of fox and rabbit populations. """
    return array([ a*X[0] -   b*X[0]*X[1] ,
                  -c*X[1] + d*b*X[0]*X[1] ])
```

Population equilibrium

Before using !SciPy to integrate this system, we will have a closer look at position equilibrium. Equilibrium occurs when the growth rate is equal to 0. This gives two fixed points:

```
#!python
X_f0 = array([     0. ,  0.])
X_f1 = array([ c/(d*b), a/b])
all(dX_dt(X_f0) == zeros(2) ) and all(dX_dt(X_f1) == zeros(2)) # =>
```

Stability of the fixed points

Near these two points, the system can be linearized: dX_dt = A_f*X where A is the Jacobian matrix evaluated at the corresponding point. We have to define the Jacobian matrix:

```
#!python
def d2X_dt2(X, t=0):
    """ Return the Jacobian matrix evaluated in X. """
    return array([[a -b*X[1],   -b*X[0]     ],
                  [b*d*X[1] ,   -c +b*d*X[0]] ])
```

So near X_f0, which represents the extinction of both species, we have:

```
#! python
A_f0 = d2X_dt2(X_f0)                    # >>> array([[ 1\. , -0\. ]
                                        #            [ 0\. , -1.5]]
```

Near X_f0, the number of rabbits increase and the population of foxes decrease. The origin is therefore a saddle point.

Near X_f1, we have:

```
#!python
A_f1 = d2X_dt2(X_f1)                    # >>> array([[ 0\.   , -2\.
                                        #            [ 0.75,  0\.
# whose eigenvalues are +/- sqrt(c*a).j:
lambda1, lambda2 = linalg.eigvals(A_f1) # >>> (1.22474j, -1.22474j
# They are imaginary numbers. The fox and rabbit populations are pe
# analysis. Their period is given by:
T_f1 = 2*pi/abs(lambda1)                # >>> 5.130199
```

Integrating the ODE using scipy.integrate

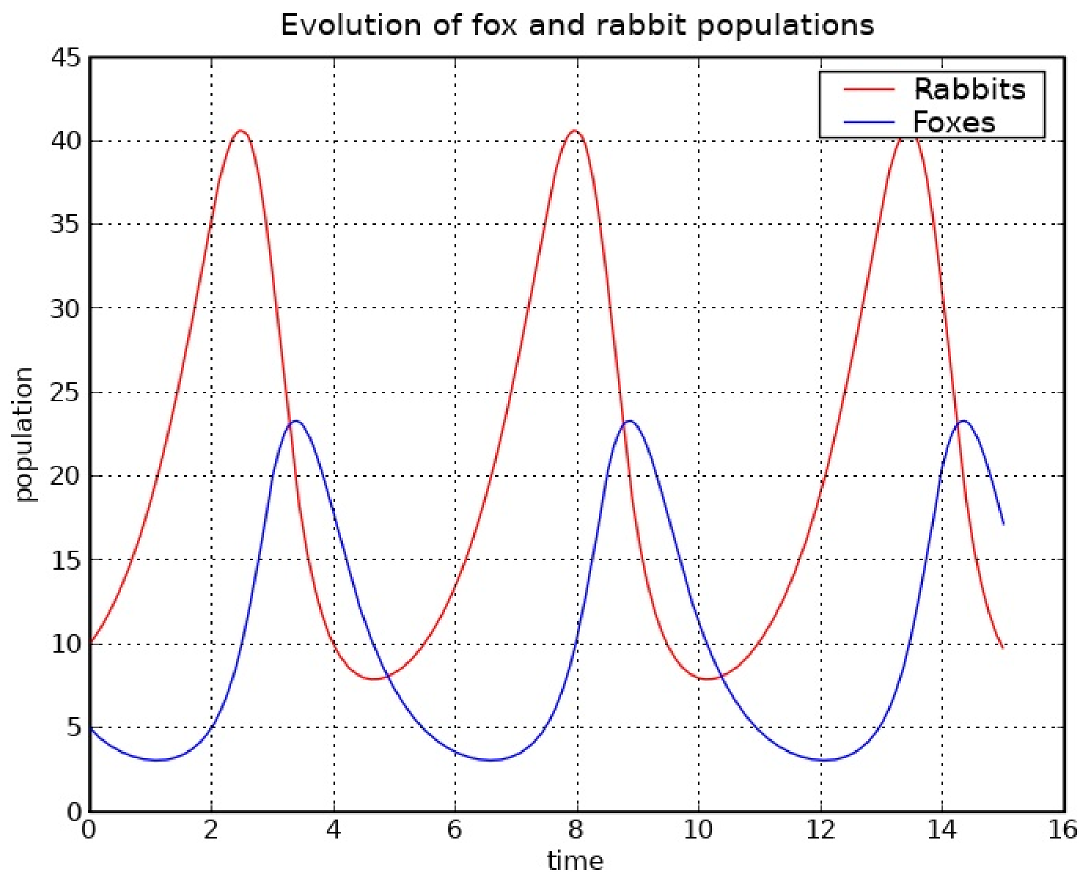
Now we will use the scipy.integrate module to integrate the ODEs. This module offers a method named odeint, which is very easy to use to integrate ODEs:

```
#!python
from scipy import integrate
t = linspace(0, 15,  1000)              # time
X0 = array([10, 5])                     # initials conditions: 10 r
X, infodict = integrate.odeint(dX_dt, X0, t, full_output=True)
infodict['message']                     # >>> 'Integration successf
```

`infodict` is optional, and you can omit the `full_output` argument if you don't want it. Type "info(odeint)" if you want more information about odeint inputs and outputs.

We can now use Matplotlib to plot the evolution of both populations:

```
#!python
rabbits, foxes = X.T
f1 = p.figure()
p.plot(t, rabbits, 'r-', label='Rabbits')
p.plot(t, foxes  , 'b-', label='Foxes')
p.grid()
p.legend(loc='best')
p.xlabel('time')
p.ylabel('population')
p.title('Evolution of fox and rabbit populations')
f1.savefig('rabbits_and_foxes_1.png')
```



The populations are indeed periodic, and their period is close to the value T_f1 that we computed.

Plotting direction fields and trajectories in the phase plane

We will plot some trajectories in a phase plane for different starting points between X_f0 and X_f1.

We will use Matplotlib's colormap to define colors for the trajectories. These colormaps are very useful to make nice plots. Have a look at ShowColormaps if you want more information.

```
values  = linspace(0.3, 0.9, 5)                          # positior
vcolors = p.cm.autumn_r(linspace(0.3, 1., len(values)))  # colors f

f2 = p.figure()

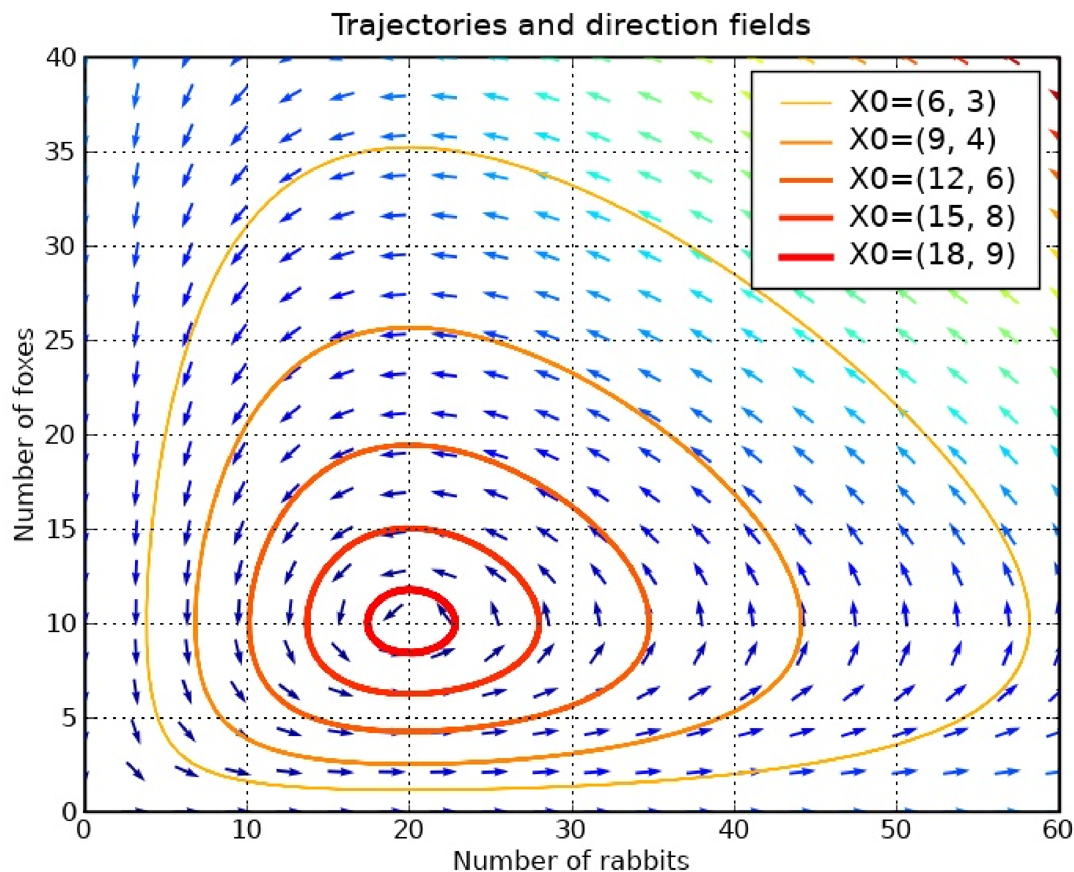
#-------------------------------------------------------
# plot trajectories
for v, col in zip(values, vcolors):
    X0 = v * X_f1                               # starting point
    X = integrate.odeint( dX_dt, X0, t)         # we don't need inf
    p.plot( X[:,0], X[:,1], lw=3.5*v, color=col, label='X0=(%.f, %.

#-------------------------------------------------------
# define a grid and compute direction at each point
ymax = p.ylim(ymin=0)[1]                        # get axis limits
xmax = p.xlim(xmin=0)[1]
nb_points   = 20

x = linspace(0, xmax, nb_points)
y = linspace(0, ymax, nb_points)

X1 , Y1  = meshgrid(x, y)                       # create a grid
DX1, DY1 = dX_dt([X1, Y1])                      # compute growth ra
M = (hypot(DX1, DY1))                           # Norm of the growt
M[ M == 0] = 1.                                 # Avoid zero divisi
DX1 /= M                                        # Normalize each ar
DY1 /= M

#-------------------------------------------------------
# Drow direction fields, using matplotlib 's quiver function
# I choose to plot normalized arrows and to use colors to give info
# the growth speed
p.title('Trajectories and direction fields')
Q = p.quiver(X1, Y1, DX1, DY1, M, pivot='mid', cmap=p.cm.jet)
p.xlabel('Number of rabbits')
p.ylabel('Number of foxes')
p.legend()
p.grid()
p.xlim(0, xmax)
p.ylim(0, ymax)
f2.savefig('rabbits_and_foxes_2.png')
```



This graph shows us that changing either the fox or the rabbit population can have an unintuitive effect. If, in order to decrease the number of rabbits, we introduce foxes, this can lead to an increase of rabbits in the long run, depending on the time of intervention.

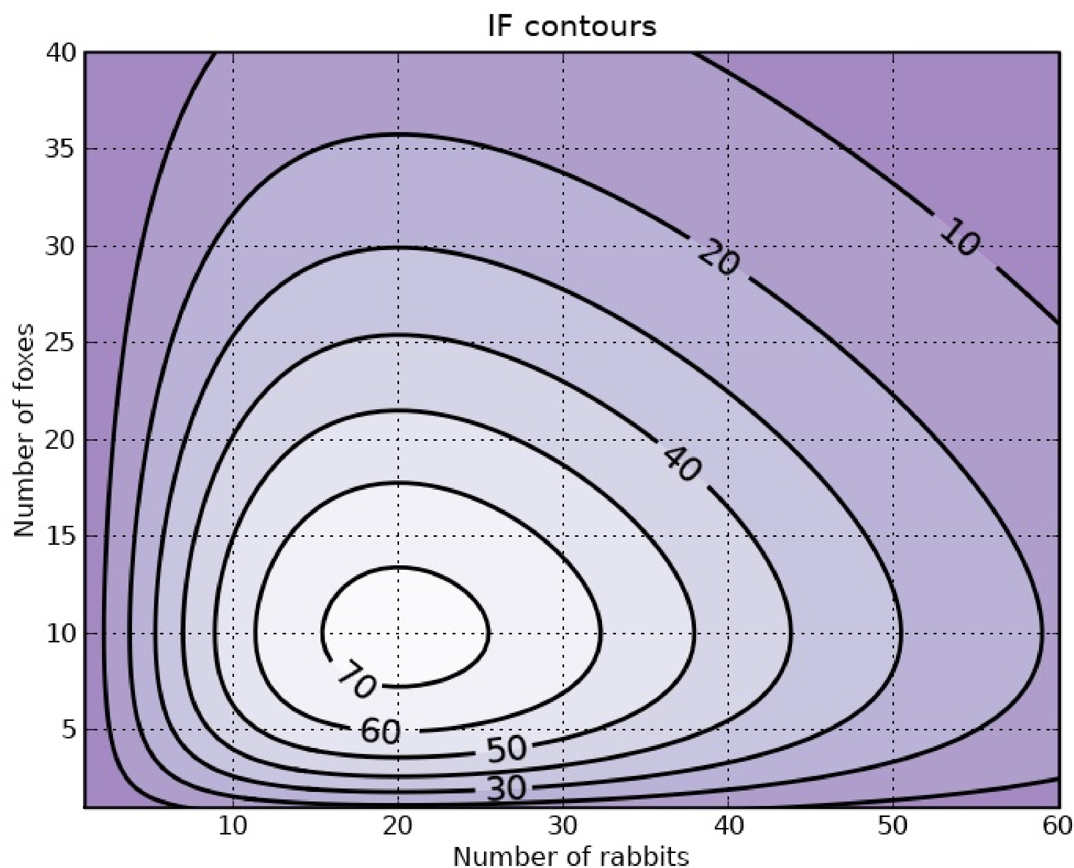
Plotting contours

We can verify that the function IF defined below remains constant along a trajectory:

```
#!python
def IF(X):
    u, v = X
    return u**(c/a) * v * exp( -(b/a)*(d*u+v) )
# We will verify that IF remains constant for different trajectorie
for v in values:
    X0 = v * X_f1                               # starting point
    X = integrate.odeint( dX_dt, X0, t)
    I = IF(X.T)                                 # compute IF along
    I_mean = I.mean()
    delta = 100 * (I.max()-I.min())/I_mean
    print 'X0=(%2.f,%2.f) => I ~ %.1f |delta = %.3G  %%' % (X0[0],
# >>> X0=( 6, 3) => I ~ 20.8 |delta = 6.19E-05 %
#     X0=( 9, 4) => I ~ 39.4 |delta = 2.67E-05 %
#     X0=(12, 6) => I ~ 55.7 |delta = 1.82E-05 %
#     X0=(15, 8) => I ~ 66.8 |delta = 1.12E-05 %
#     X0=(18, 9) => I ~ 72.4 |delta = 4.68E-06 %
```

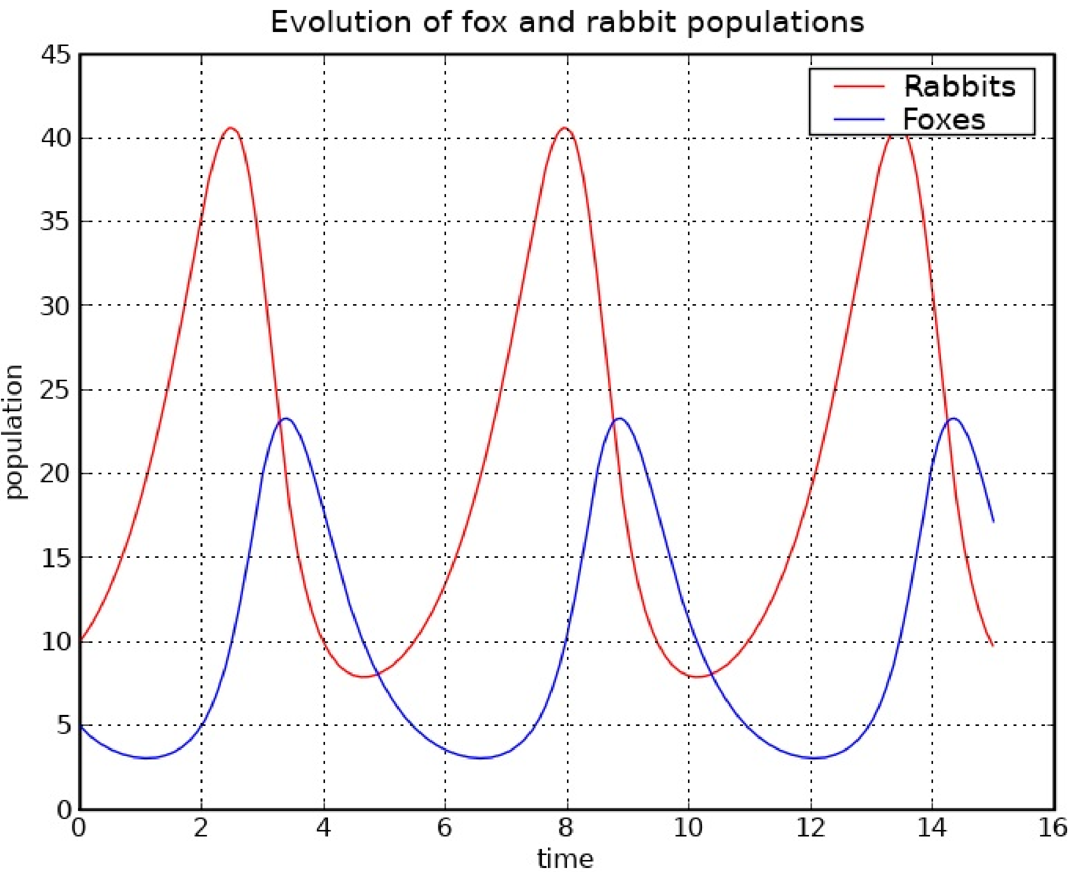
Plotting iso-contours of IF can be a good representation of trajectories, without having to integrate the ODE

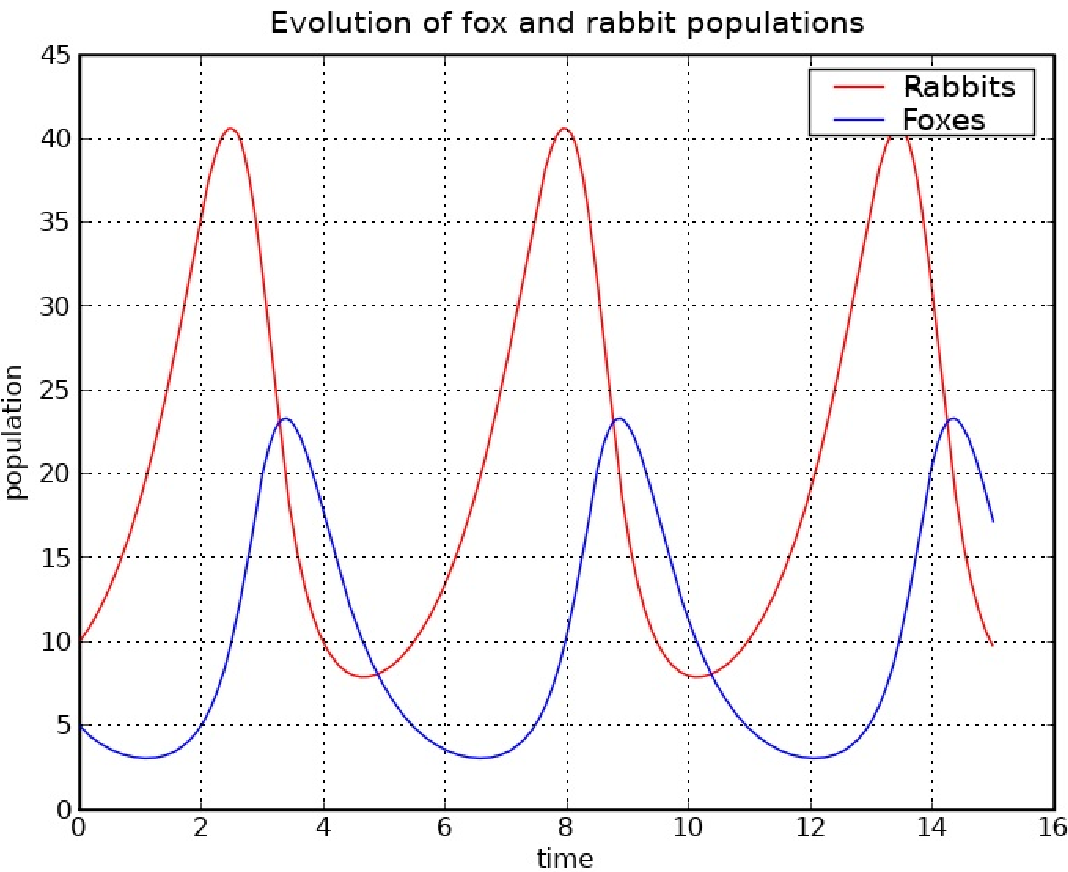
```
#!python
#-------------------------------------------------------
# plot iso contours
nb_points = 80                              # grid size
x = linspace(0, xmax, nb_points)
y = linspace(0, ymax, nb_points)
X2 , Y2  = meshgrid(x, y)                   # create the grid
Z2 = IF([X2, Y2])                           # compute IF on each po
f3 = p.figure()
CS = p.contourf(X2, Y2, Z2, cmap=p.cm.Purples_r, alpha=0.5)
CS2 = p.contour(X2, Y2, Z2, colors='black', linewidths=2. )
p.clabel(CS2, inline=1, fontsize=16, fmt='%.f')
p.grid()
p.xlabel('Number of rabbits')
p.ylabel('Number of foxes')
p.ylim(1, ymax)
p.xlim(1, xmax)
p.title('IF contours')
f3.savefig('rabbits_and_foxes_3.png')
p.show()
```



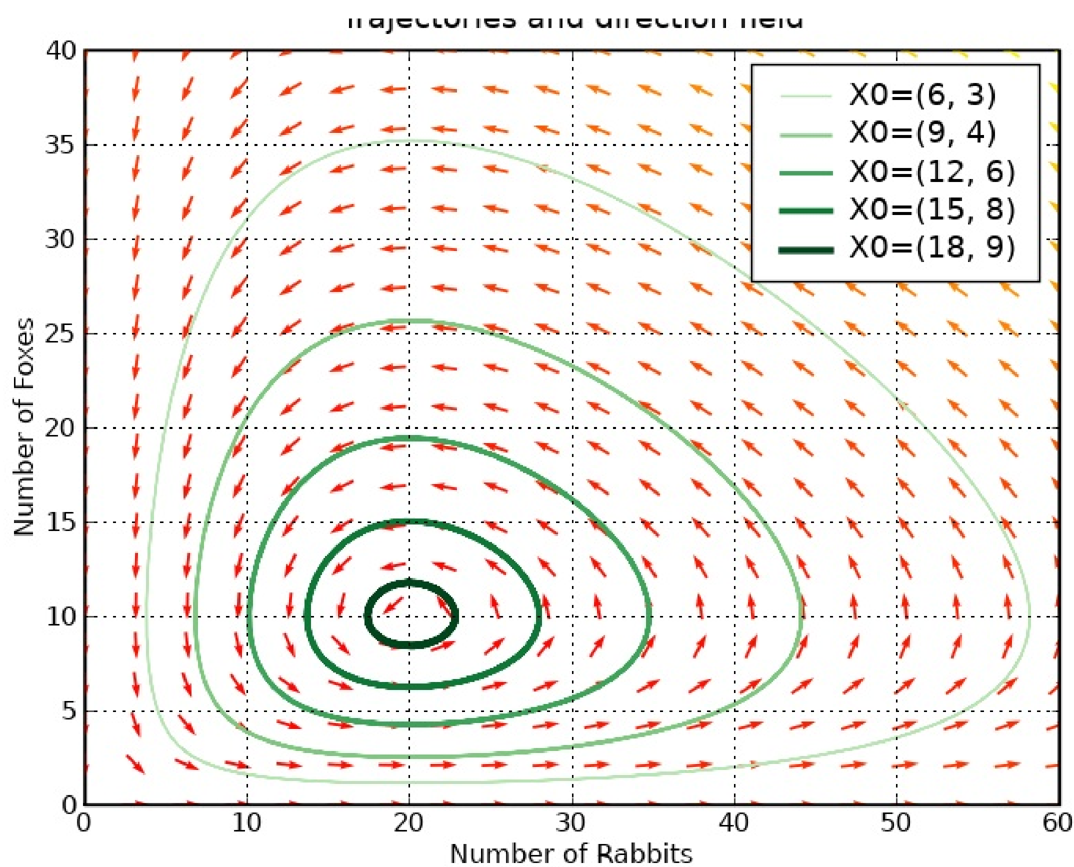
Attachments

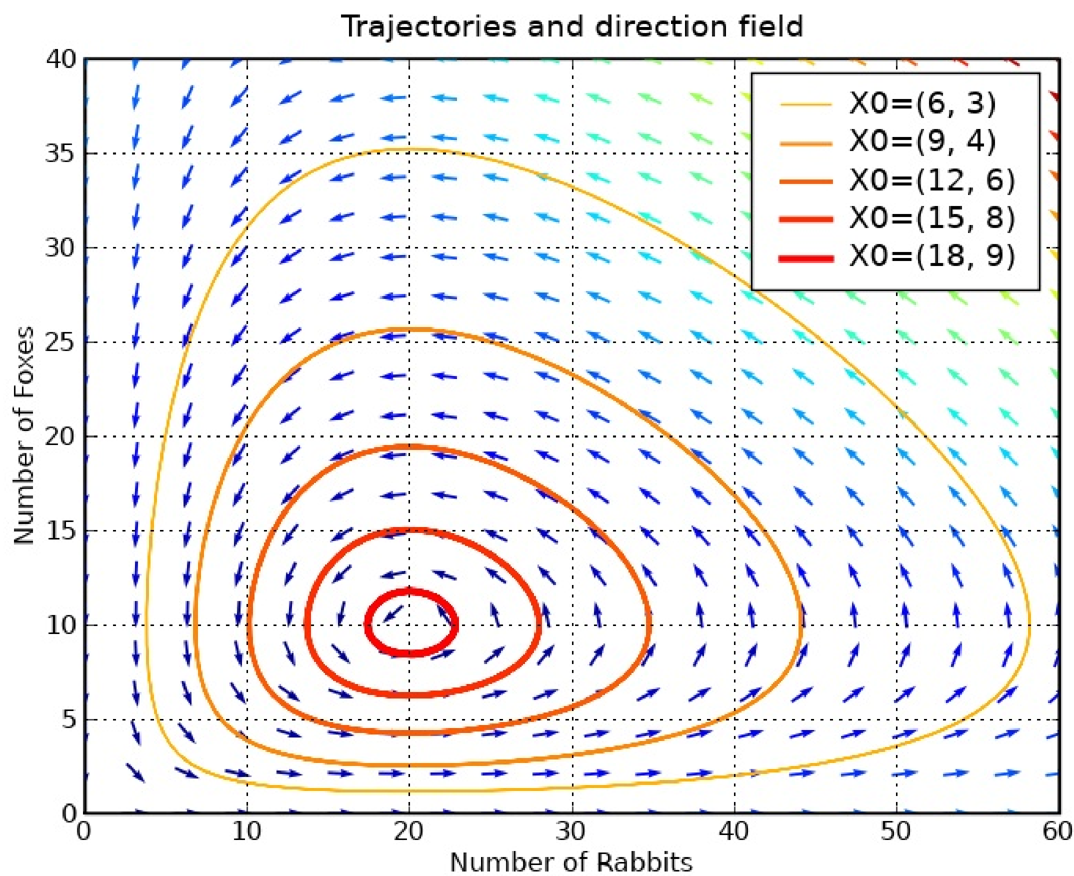
- `rabbits_and_foxes_1.png`
- `rabbits_and_foxes_1v2.png`
- `rabbits_and_foxes_2.png`
- `rabbits_and_foxes_2v2.png`
- `rabbits_and_foxes_2v3.png`
- `rabbits_and_foxes_3.png`
- `rabbits_and_foxes_3v2.png`
- `tutorial_lokta-voltera.py`
- `tutorial_lokta-voltera_v2.py`
- `tutorial_lokta-voltera_v3.py`
- `tutorial_lokta-voltera_v4.py`
- `zombie_nodead_nobirths.png`
- `zombie_somedead_10births.png`
- `zombie_somedead_nobirths.png`
- `zombie_somedeaddead_nobirths.png`

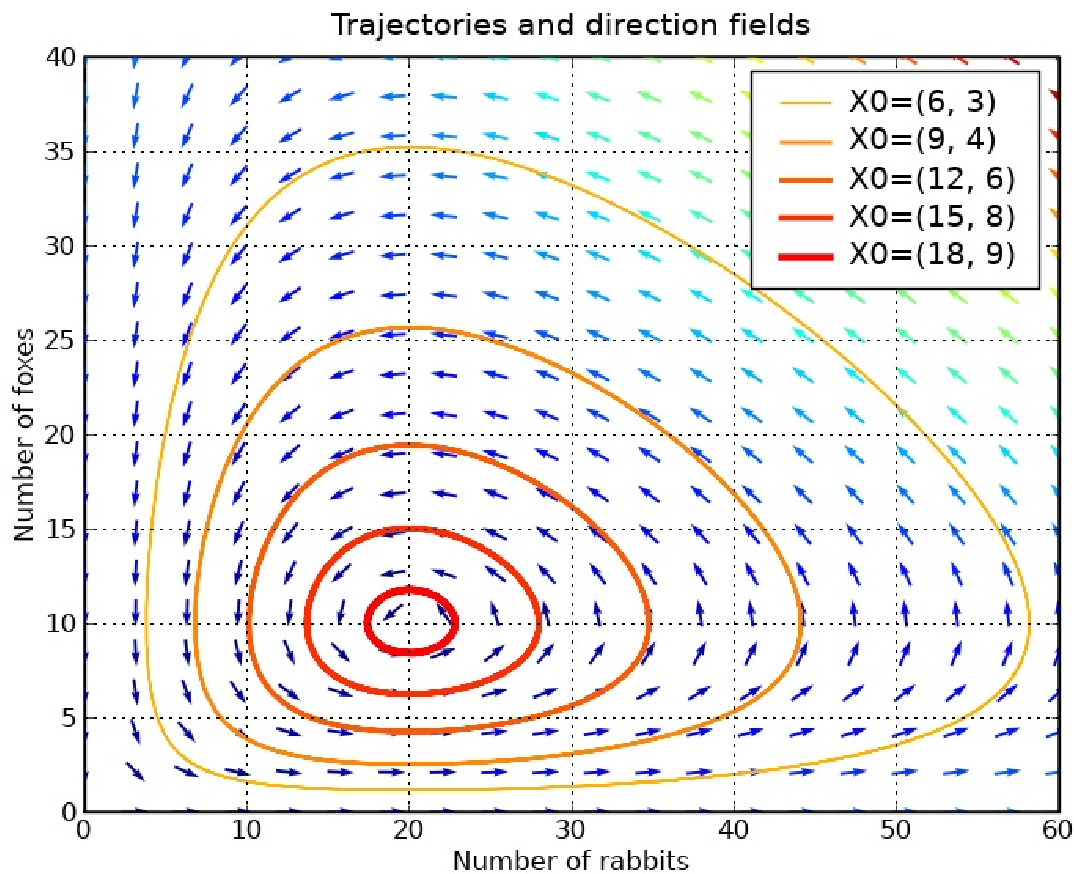


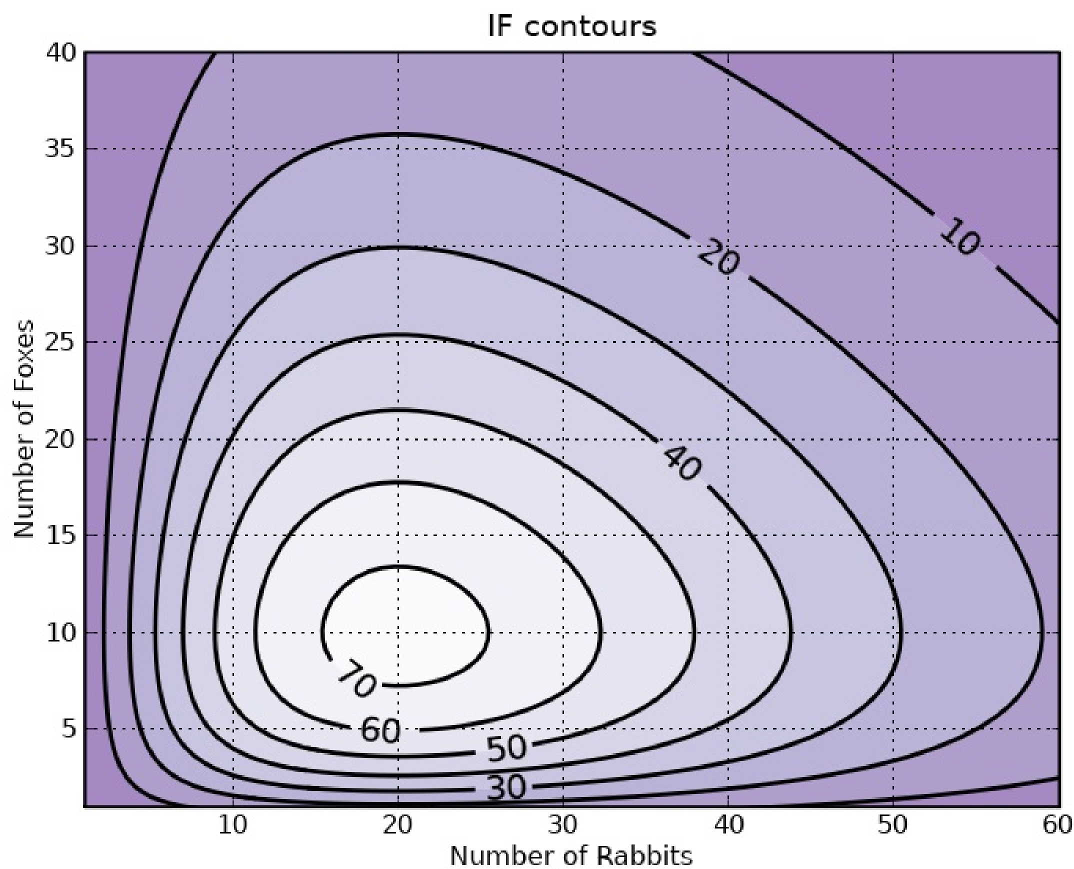


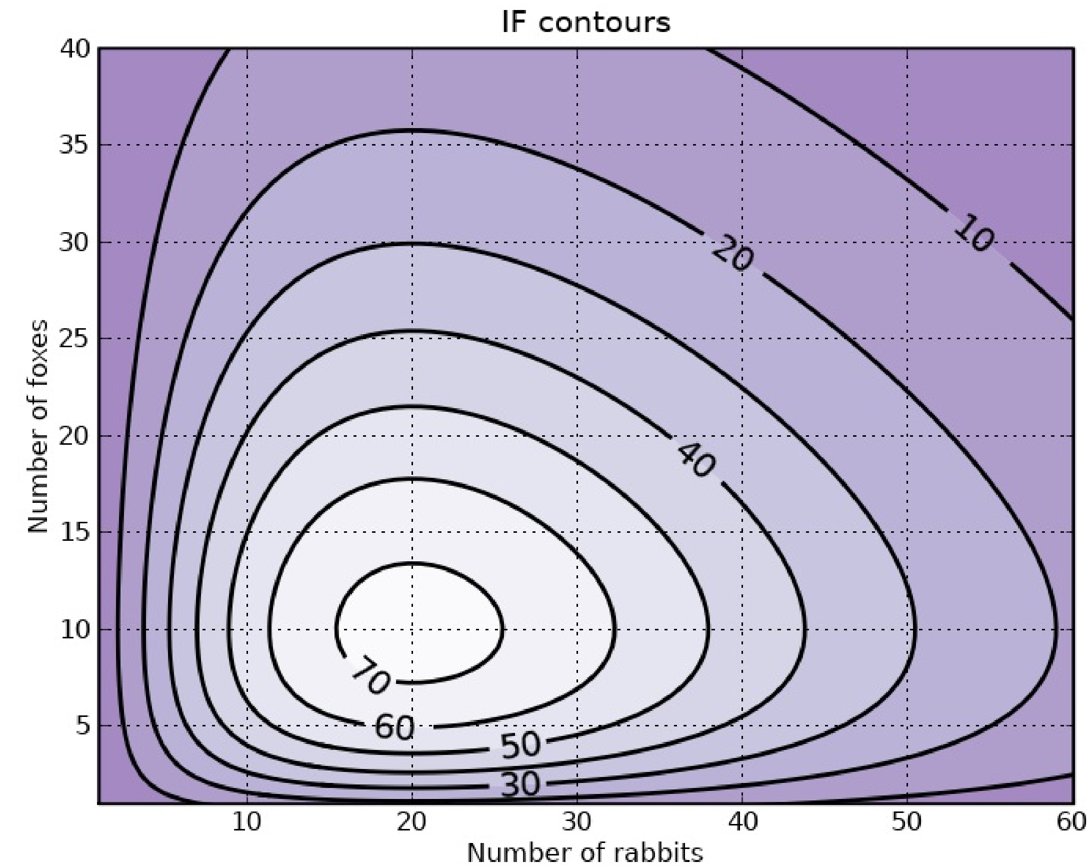
Trajectories and direction field

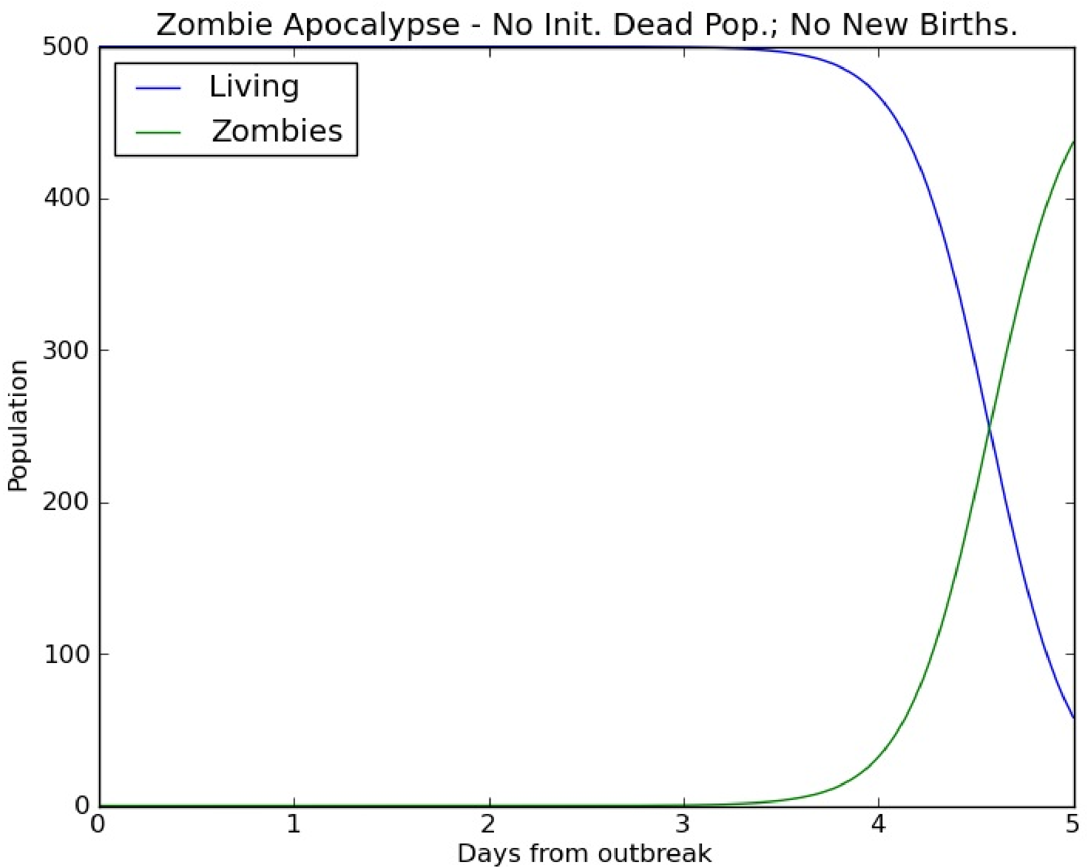

Trajectories and direction field
X0=(6, 3)
X0=(9, 4)
X0=(12, 6)
X0=(15, 8)
X0=(18, 9)
Number of Foxes
Number of Rabbits
0
10
20
30
40
50
60
5
15
25
35

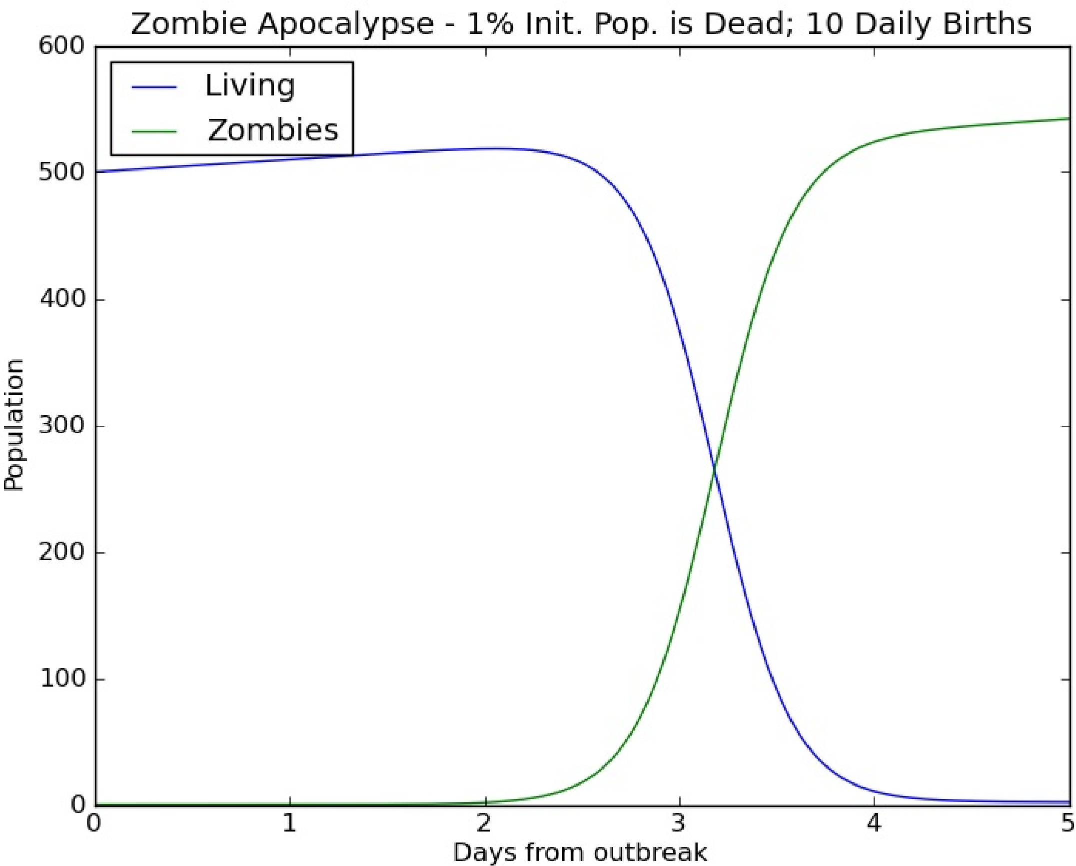

Trajectories and direction field
X0=(6, 3)
X0=(9, 4)
X0=(12, 6)
X0=(15, 8)
X0=(18, 9)
Number of Foxes
Number of Rabbits
0
10
20
30
40
50
60
5
15
25
35

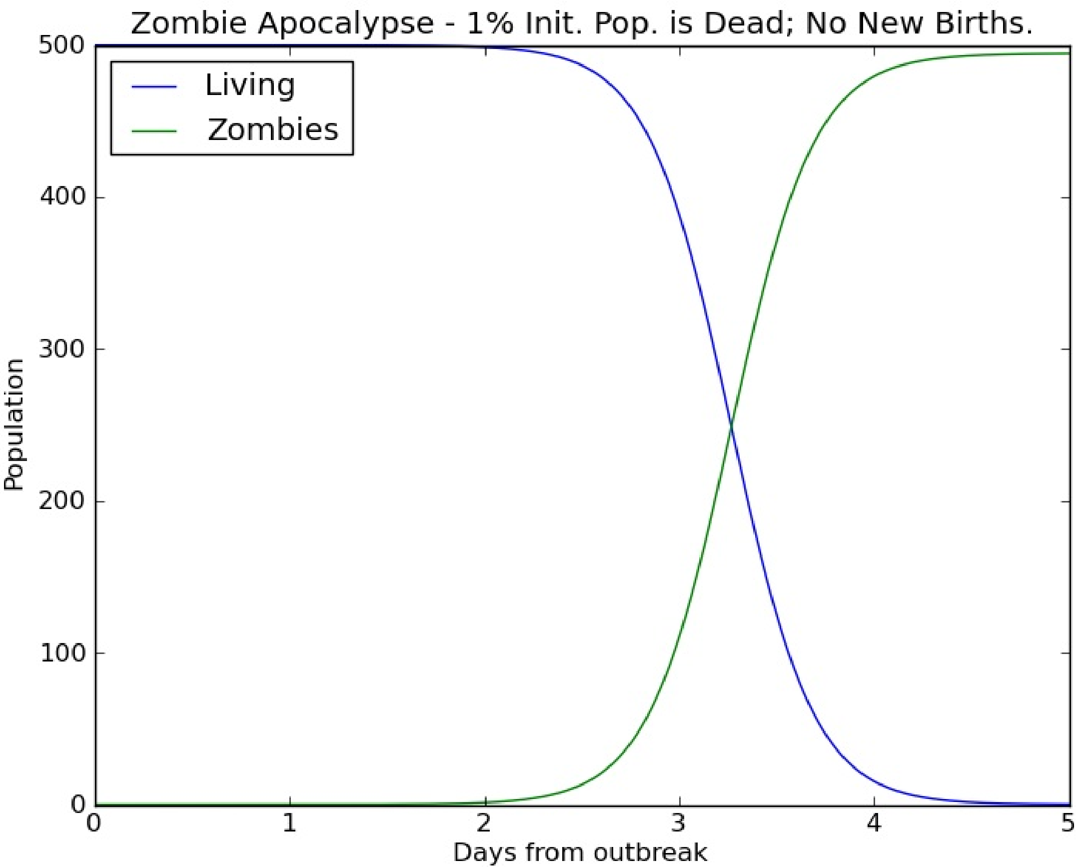

Trajectories and direction fields
X0=(6, 3)
X0=(9, 4)
X0=(12, 6)
X0=(15, 8)
X0=(18, 9)
Number of foxes
Number of rabbits
0
5
10
15
20
25
30
35
40
0
10
20
30
40
50
60

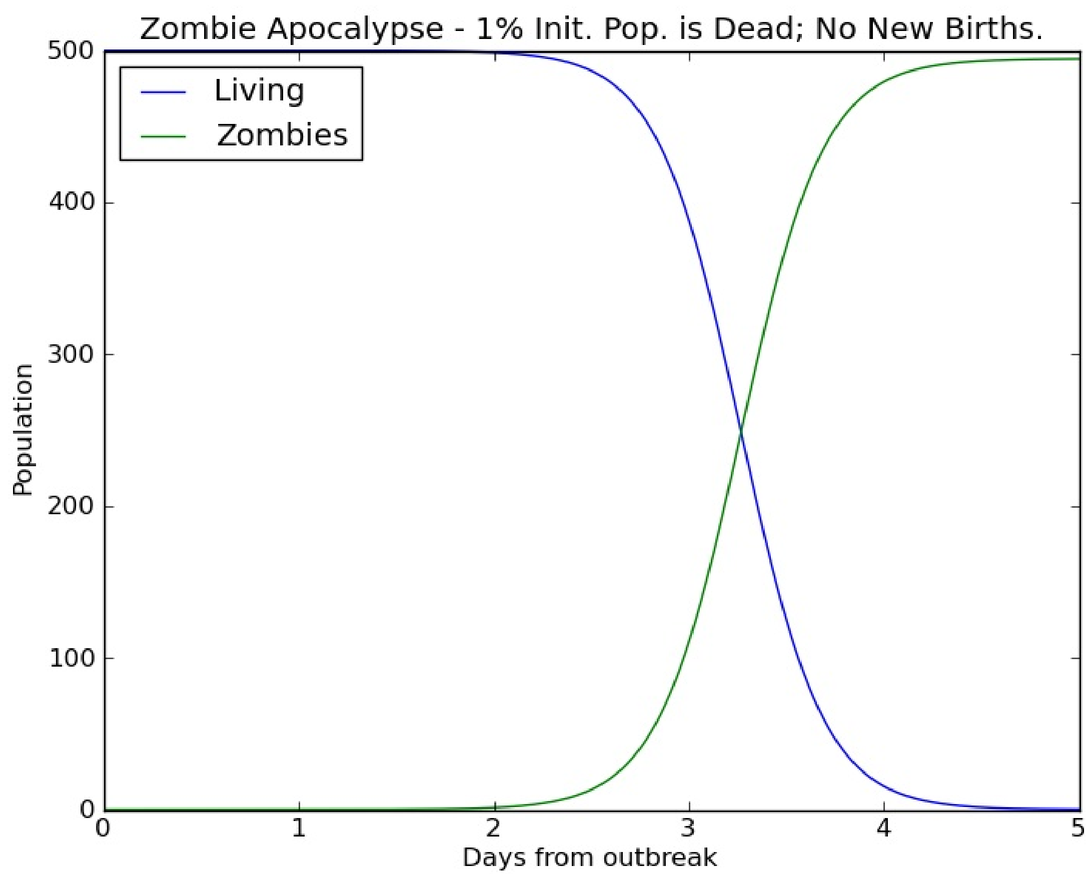

IF contours
Number of Foxes
Number of Rabbits
10
20
30
40
50
60
70
5
10
15
20
25
30
35
40
10
20
30
40
50
60


IF contours
Number of foxes
Number of rabbits
40
35
30
25
20
15
10
5
10
20
30
40
50
60
70
60
50
40
30
20
10


Zombie Apocalypse - No Init. Dead Pop.; No New Births.
Living
Zombies
Population
Days from outbreak
500
400
300
200
100
0
0
1
2
3
4
5


Zombie Apocalypse - 1% Init. Pop. is Dead; 10 Daily Births
Living
Zombies
Population
Days from outbreak
0
100
200
300
400
500
600
0
1
2
3
4
5


Zombie Apocalypse - 1% Init. Pop. is Dead; No New Births.
Living
Zombies
Population
Days from outbreak
0
100
200
300
400
500
0
1
2
3
4
5


Zombie Apocalypse - 1% Init. Pop. is Dead; No New Births.
Living
Zombies
Population
Days from outbreak
0
100
200
300
400
500
0
1
2
3
4
5

Modeling a Zombie Apocalypse

This example demonstrates how to solve a system of first order ODEs using SciPy. Note that a Nth order equation can also be solved using SciPy by transforming it into a system of first order equations. In a this lighthearted example, a system of ODEs can be used to model a “zombie invasion”, using the equations specified in Munz et al. 2009.

The system is given as:

dS/dt = P - B_S_Z - d_S dZ/dt = B_S_Z + G_R - A_S_Z dR/dt = d_S + A_S_Z - G_R

with the following notations:

- S: the number of susceptible victims
- Z: the number of zombies
- R: the number of people “killed”
- P: the population birth rate
- d: the chance of a natural death
- B: the chance the “zombie disease” is transmitted (an alive person becomes a zombie)
- G: the chance a dead person is resurrected into a zombie
- A: the chance a zombie is totally destroyed

This involves solving a system of first order ODEs given by: d**y**/dt = **f**(**y**, t)

Where **y** = [S, Z, R].

The code used to solve this system is below:

```
# zombie apocalypse modeling
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
plt.ion()
plt.rcParams['figure.figsize'] = 10, 8

P = 0           # birth rate
d = 0.0001  # natural death percent (per day)
B = 0.0095  # transmission percent  (per day)
G = 0.0001  # resurect percent (per day)
A = 0.0001  # destroy percent  (per day)

# solve the system dy/dt = f(y, t)
def f(y, t):
    Si = y[0]
    Zi = y[1]
    Ri = y[2]
```

```
    # the model equations (see Munz et al. 2009)
    f0 = P - B*Si*Zi - d*Si
    f1 = B*Si*Zi + G*Ri - A*Si*Zi
    f2 = d*Si + A*Si*Zi - G*Ri
    return [f0, f1, f2]

# initial conditions
S0 = 500.                   # initial population
Z0 = 0                      # initial zombie population
R0 = 0                      # initial death population
y0 = [S0, Z0, R0]   # initial condition vector
t  = np.linspace(0, 5., 1000)       # time grid

# solve the DEs
soln = odeint(f, y0, t)
S = soln[:, 0]
Z = soln[:, 1]
R = soln[:, 2]

# plot results
plt.figure()
plt.plot(t, S, label='Living')
plt.plot(t, Z, label='Zombies')
plt.xlabel('Days from outbreak')
plt.ylabel('Population')
plt.title('Zombie Apocalypse - No Init. Dead Pop.; No New Births.')
plt.legend(loc=0)

# change the initial conditions
R0 = 0.01*S0   # 1% of initial pop is dead
y0 = [S0, Z0, R0]

# solve the DEs
soln = odeint(f, y0, t)
S = soln[:, 0]
Z = soln[:, 1]
R = soln[:, 2]

plt.figure()
plt.plot(t, S, label='Living')
plt.plot(t, Z, label='Zombies')
plt.xlabel('Days from outbreak')
plt.ylabel('Population')
plt.title('Zombie Apocalypse - 1% Init. Pop. is Dead; No New Births
plt.legend(loc=0)

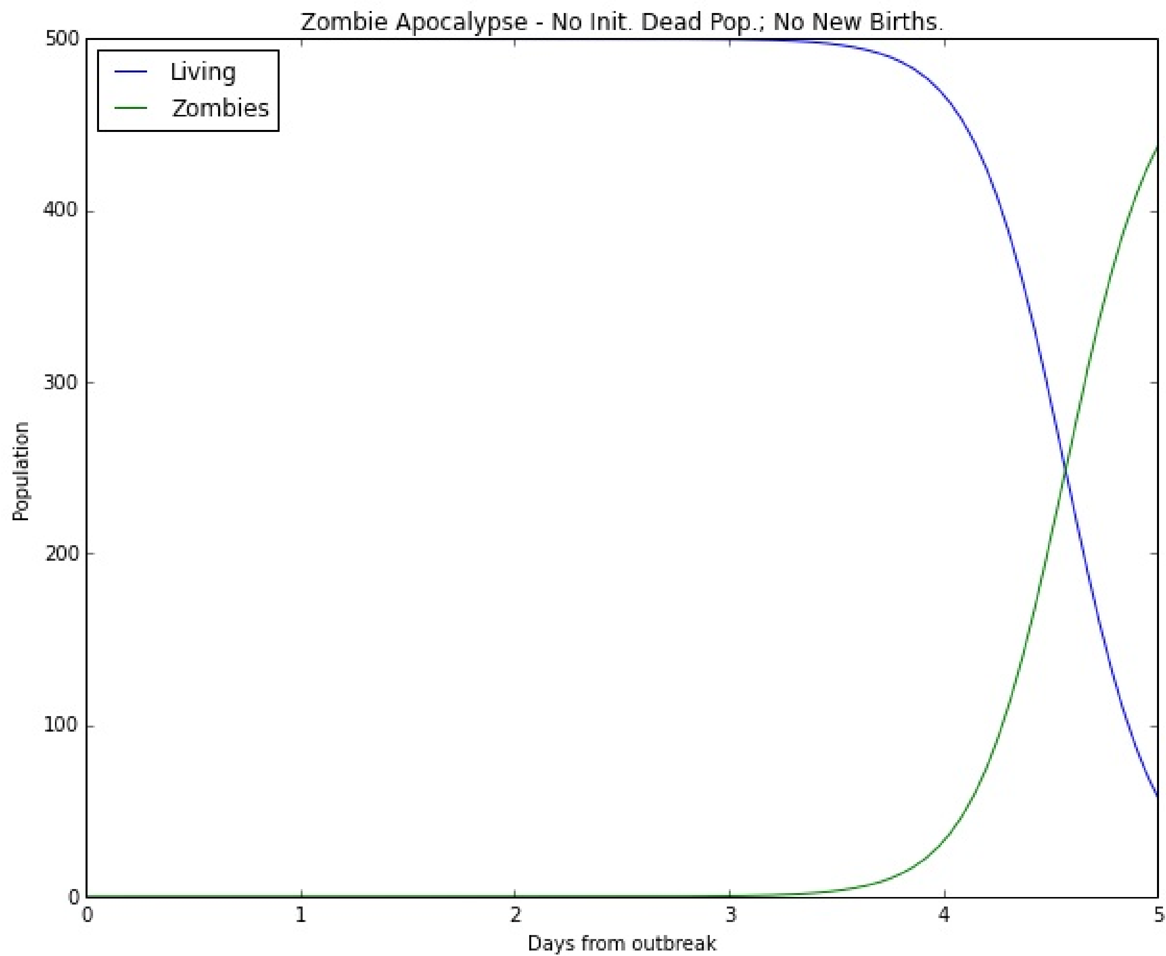
# change the initial conditions
R0 = 0.01*S0   # 1% of initial pop is dead
P  = 10        # 10 new births daily
y0 = [S0, Z0, R0]

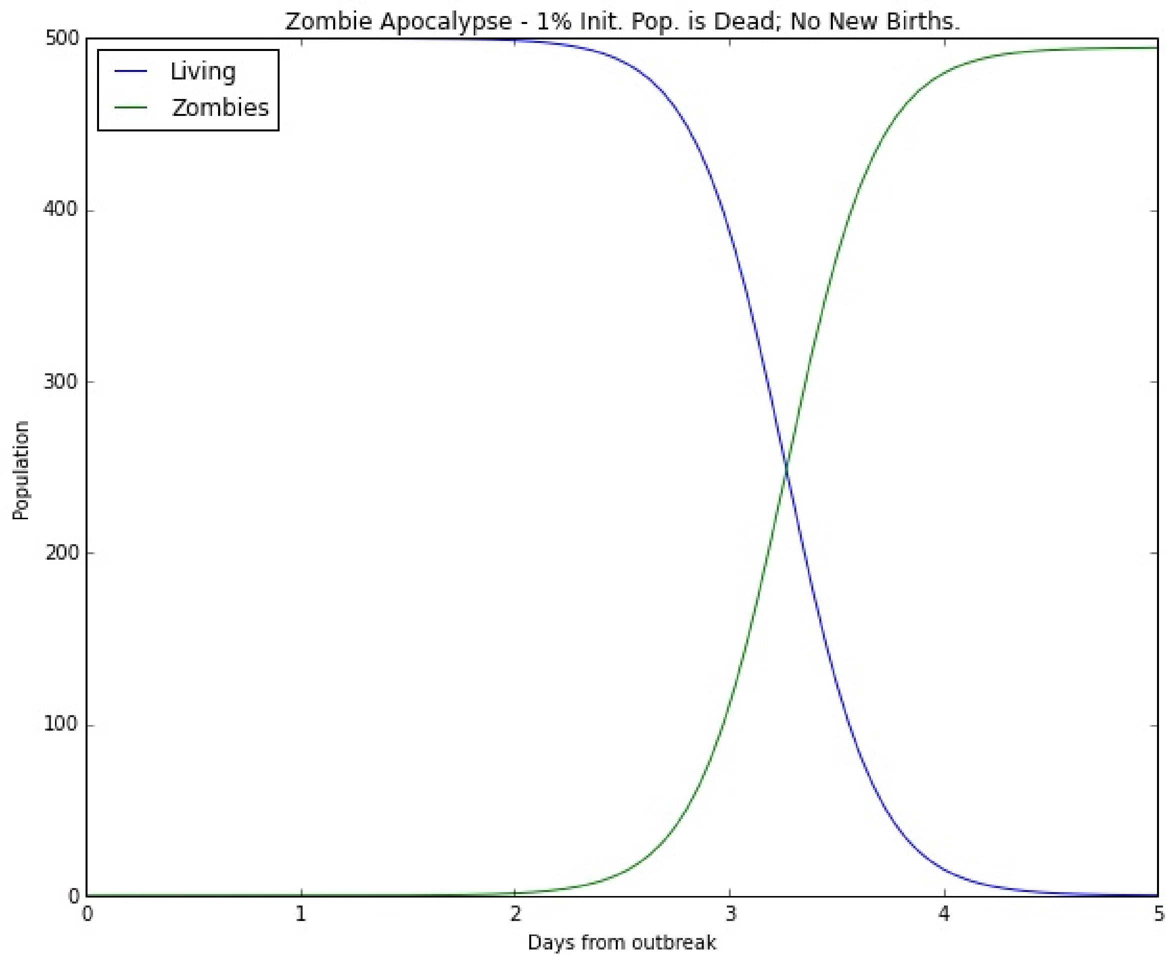
# solve the DEs
soln = odeint(f, y0, t)
```

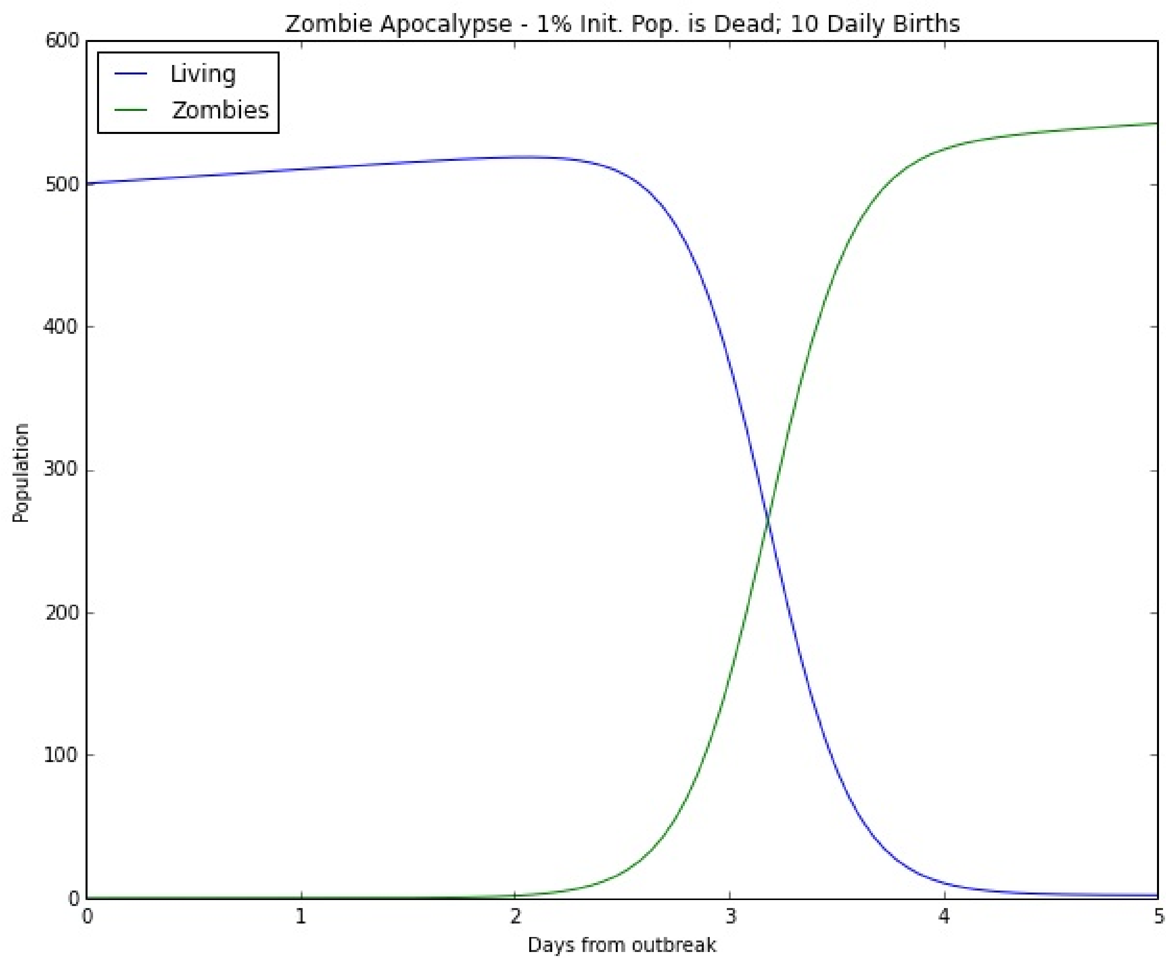
```
S = soln[:, 0]
Z = soln[:, 1]
R = soln[:, 2]

plt.figure()
plt.plot(t, S, label='Living')
plt.plot(t, Z, label='Zombies')
plt.xlabel('Days from outbreak')
plt.ylabel('Population')
plt.title('Zombie Apocalypse - 1% Init. Pop. is Dead; 10 Daily Birt
plt.legend(loc=0)
```

```
<matplotlib.legend.Legend at 0x392ac90>
```


Zombie Apocalypse - No Init. Dead Pop.; No New Births.
Living
Zombies
Population
Days from outbreak
0
100
200
300
400
500
0
1
2
3
4
5


Zombie Apocalypse - 1% Init. Pop. is Dead; No New Births.
Living
Zombies
Population
Days from outbreak
0
100
200
300
400
500
0
1
2
3
4
5


Zombie Apocalypse - 1% Init. Pop. is Dead; 10 Daily Births
Living
Zombies
Population
Days from outbreak
0
100
200
300
400
500
600
0
1
2
3
4
5

Theoretical ecology: Hastings and Powell

Overview

A simple script that recreates the min/max bifurcation diagrams from Hastings and Powell 1991.

Library Functions

Two useful functions are defined in the module bif.py.

```
import numpy

def window(data, size):
    """A generator that returns the moving window of length
 `size` over the `data`

 """
    for start in range(len(data) - (size - 1)):
        yield data[start:(start + size)]

def min_max(data, tol=1e-14):
    """Return a list of the local min/max found
 in a `data` series, given the relative tolerance `tol`

 """
    maxes = []
    mins = []
    for first, second, third in window(data, size=3):
        if first < second and third < second:
            maxes.append(second)
        elif first > second and third > second:
            mins.append(second)
        elif abs(first - second) < tol and abs(second - third) < to
            # an equilibrium is both the maximum and minimum
            maxes.append(second)
            mins.append(second)

    return {'max': numpy.asarray(maxes),
            'min': numpy.asarray(mins)}
```

The Model

For speed the model is defined in a fortran file and compiled into a library for use from python. Using this method gives a 100 fold increase in speed. The file uses Fortran 90, which makes using f2py especially easy. The file is named hastings.f90.

```
module model
    implicit none

    real(8), save :: a1, a2, b1, b2, d1, d2

contains

    subroutine fweb(y, t, yprime)
        real(8), dimension(3), intent(in) :: y
        real(8), intent(in) :: t
        real(8), dimension(3), intent(out) :: yprime

        yprime(1) = y(1)*(1.0d0 - y(1)) - a1*y(1)*y(2)/(1.0d0 + b1*
        yprime(2) = a1*y(1)*y(2)/(1.0d0 + b1*y(1)) - a2*y(2)*y(3)/(
        yprime(3) = a2*y(2)*y(3)/(1.0d0 + b2*y(2)) - d2*y(3)
    end subroutine fweb

end module model
```

Which is compiled (using the gfortran compiler) with the command: `f2py -c -m hastings hastings.f90 --fcompiler=gnu95`

```
import numpy
from scipy.integrate import odeint
import bif

import hastings

# setup the food web parameters
hastings.model.a1 = 5.0
hastings.model.a2 = 0.1
hastings.model.b2 = 2.0
hastings.model.d1 = 0.4
hastings.model.d2 = 0.01

# setup the ode solver parameters
t = numpy.arange(10000)
y0 = [0.8, 0.2, 10.0]

def print_max(data, maxfile):
    for a_max in data['max']:
        print >> maxfile, hastings.model.b1, a_max

x_maxfile = open('x_maxfile.dat', 'w')
y_maxfile = open('y_maxfile.dat', 'w')
z_maxfile = open('z_maxfile.dat', 'w')
for i, hastings.model.b1 in enumerate(numpy.linspace(2.0, 6.2, 420)
    print i, hastings.model.b1
    y = odeint(hastings.model.fweb, y0, t)

    # use the last 'stationary' solution as an intial guess for the
    # next run. This both speeds up the computations, as well as he
    # make sure that solver doesn't need to do too much work.
    y0 = y[-1, :]

    x_minmax = bif.min_max(y[5000:, 0])
    y_minmax = bif.min_max(y[5000:, 1])
    z_minmax = bif.min_max(y[5000:, 2])

    print_max(x_minmax, x_maxfile)
    print_max(y_minmax, y_maxfile)
    print_max(z_minmax, z_maxfile)
```

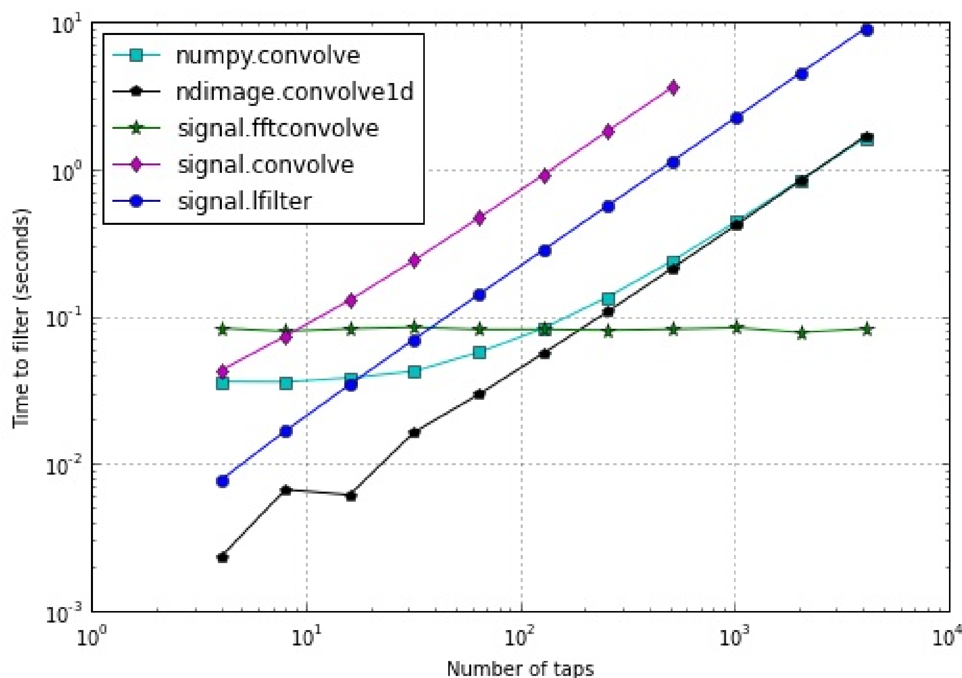
Numpy & Scipy / Other examples

Applying a FIR filter

How to apply a FIR filter: convolve, fftconvolve, convolve1d or lfilter?

The following plot shows the time required to apply a finite impulse response (FIR) filter of varying length to a signal of length 131072 using several different functions that are available in numpy and scipy. The details of how this figure was created are given below.

```
fig
```



The Details

There are several functions in the numpy and scipy libraries that can be used to apply a FIR filter to a signal. From scipy.signal, lfilter() is designed to apply a discrete IIR filter to a signal, so by simply setting the array of denominator coefficients to [1.0], it can be used to apply a FIR filter. Applying a FIR filter is equivalent to a discrete convolution, so one can also use convolve() from numpy, convolve() or fftconvolve() from scipy.signal, or convolve1d() from scipy.ndimage. In this page, we demonstrate each of these functions, and we look at how the

computational time varies when the data signal size is fixed and the FIR filter length is varied. We'll use a data signal length of 131072, which is 2**17. We assume that we have *m* channels of data, so our input signal is an *m* by *n* array.

We assume our FIR filter coefficients are in a one-dimensional array *b*. The function numpy.convolve only accepts one-dimensional arrays, so we'll have to use a python loop over our input array to perform the convolution over all the channels. One way to do that is

```
y = np.array([np.convolve(xi, b, mode='valid') for xi in x])
```

We use a list comprehension to loop over the rows of *x*, and pass the result to *np.array* to reassemble the filtered data into a two-dimensional array.

Both signal.convolve and signal.fftconvolve perform a two-dimensional convolution of two-dimensional arrays. To filter our *m* by *n* array with either of these functions, we shape our filter to be a two-dimensional array, with shape 1 by *len(b)*. The python code looks like this:

```
y = convolve(x, b[np.newaxis, :], mode='valid')
```

where *x* is a numpy array with shape *(m, n)*, and *b* is the one-dimensional array of FIR filter coefficients. *b[np.newaxis, :]* is the two dimensional view of *b*, with shape 1 by *len(b)*. *y* is the filtered data; it includes only those terms for which the full convolution was computed, so it has shape *(m, n - len(b) + 1)*.

ndimage.convolve1d() is designed to do a convolution of a 1d array along the given axis of another n-dimensional array. It does not have the option mode='valid', so to extract the valid part of the result, we slice the result of the function:

```
y = convolve1d(x, b)[:, (len(b)-1)//2 : -(len(b)//2)]
```

signal.lfilter is designed to filter one-dimensional data. It can take a two-dimensional array (or, in general, an n-dimensional array) and filter the data in any given axis. It can also be used for IIR filters, so in our case, we'll pass in [1.0] for the denominator coefficients. In python, this looks like:

```
y = lfilter(b, [1.0], x)
```

To obtain exactly the same array as computed by convolve or fftconvolve (i.e. to get the equivalent of the 'valid' mode), we must discard the beginning of the array computed by lfilter. We can do this by slicing the array immediately after the call to filter:

```
y = lfilter(b, [1.0], x)[:, len(b) - 1:]
```

The following script computes and plots the results of applying a FIR filter to a 2 by 131072 array of data, with a series of FIR filters of increasing length.

```
#!python
import time

import numpy as np
from numpy import convolve as np_convolve
from scipy.signal import convolve as sig_convolve, fftconvolve, lfi
from scipy.ndimage import convolve1d
from pylab import grid, show, legend, loglog, xlabel, ylabel, figu

# Create the m by n data to be filtered.
m = 4
n = 2 ** 17
x = np.random.random(size=(m, n))

conv_time = []
npconv_time = []
fftconv_time = []
conv1d_time = []
lfilt_time = []

diff_list = []
diff2_list = []
diff3_list = []

ntaps_list = 2 ** np.arange(2, 13)

for ntaps in ntaps_list:
    # Create a FIR filter.
    b = firwin(ntaps, [0.05, 0.95], width=0.05, pass_zero=False)

    if ntaps <= 2 ** 9:
        # --- signal.convolve ---
        # We know this is slower than the others when ntaps is
        # large, so we only compute it for small values.
        tstart = time.time()
        conv_result = sig_convolve(x, b[np.newaxis, :], mode='valid
        conv_time.append(time.time() - tstart)

    # --- numpy.convolve ---
    tstart = time.time()
    npconv_result = np.array([np_convolve(xi, b, mode='valid') for
    npconv_time.append(time.time() - tstart)

    # --- signal.fftconvolve ---
    tstart = time.time()
```

```
    fftconv_result = fftconvolve(x, b[np.newaxis, :], mode='valid')
    fftconv_time.append(time.time() - tstart)

    # --- convolve1d ---
    tstart = time.time()
    # convolve1d doesn't have a 'valid' mode, so we expliclity slic
    # the valid part of the result.
    conv1d_result = convolve1d(x, b)[:, (len(b)-1)//2 : -(len(b)//2
    conv1d_time.append(time.time() - tstart)

    # --- lfilter ---
    tstart = time.time()
    lfilt_result = lfilter(b, [1.0], x)[:, len(b) - 1:]
    lfilt_time.append(time.time() - tstart)

    diff = np.abs(fftconv_result - lfilt_result).max()
    diff_list.append(diff)

    diff2 = np.abs(conv1d_result - lfilt_result).max()
    diff2_list.append(diff2)

    diff3 = np.abs(npconv_result - lfilt_result).max()
    diff3_list.append(diff3)

# Verify that np.convolve and lfilter gave the same results.
print "Did np.convolve and lfilter produce the same results?",
check = all(diff < 1e-13 for diff in diff3_list)
if check:
    print "Yes."
else:
    print "No!  Something went wrong."

# Verify that fftconvolve and lfilter gave the same results.
print "Did fftconvolve and lfilter produce the same results?",
check = all(diff < 1e-13 for diff in diff_list)
if check:
    print "Yes."
else:
    print "No!  Something went wrong."

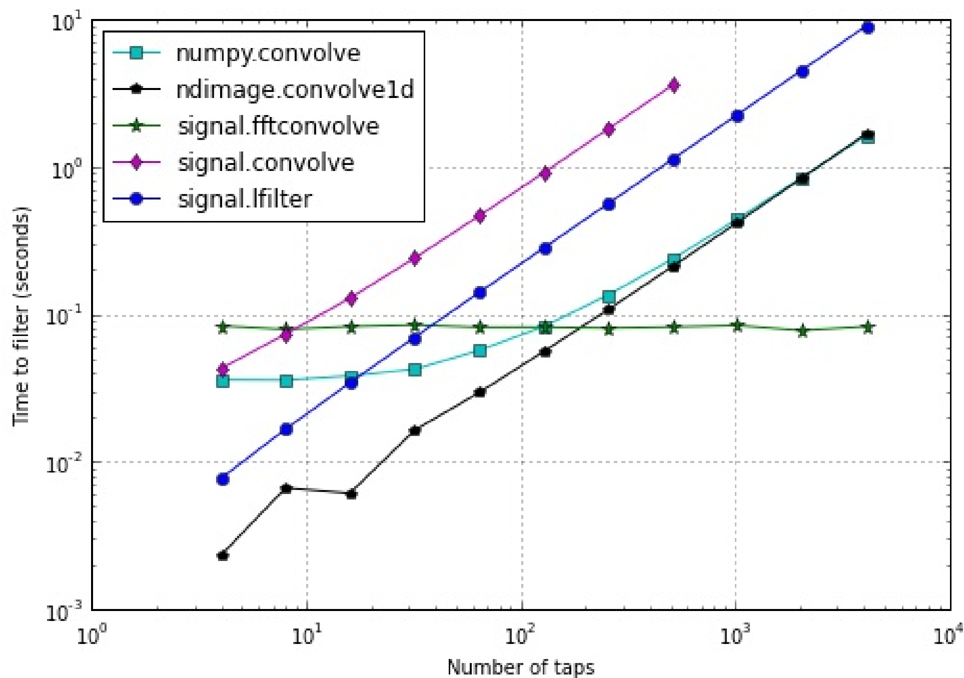
# Verify that convolve1d and lfilter gave the same results.
print "Did convolve1d and lfilter produce the same results?",
check = all(diff2 < 1e-13 for diff2 in diff2_list)
if check:
    print "Yes."
else:
    print "No!  Something went wrong."

fig = figure(1, figsize=(8, 5.5))
loglog(ntaps_list, npconv_time, 'c-s', label='numpy.convolve')
loglog(ntaps_list, conv1d_time, 'k-p', label='ndimage.convolve1d')
loglog(ntaps_list, fftconv_time, 'g-*', markersize=8, label='signal
loglog(ntaps_list[:len(conv_time)], conv_time, 'm-d', label='signal
```

```
loglog(ntaps_list, lfilt_time, 'b-o', label='signal.lfilter')
legend(loc='best', numpoints=1)
grid(True)
xlabel('Number of taps')
ylabel('Time to filter (seconds)')
```

```
Did np.convolve and lfilter produce the same results? Yes.
Did fftconvolve and lfilter produce the same results? Yes.
Did convolve1d and lfilter produce the same results? Yes.
```

```
<matplotlib.text.Text at 0x4612c50>
```



The plot shows that, depending on the number of taps, either scipy.ndimage.convolve1d, numpy.convolve or scipy.signal.fftconvolve is the fastest. The above script can be used to explore variations of these results.

Brownian Motion

Brownian motion is a stochastic process. One form of the equation for Brownian motion is

`\(X(0) = X_0\)`

`\(X(t + dt) = X(t) + N(0, (delta)^2 dt; t, t+dt)\)`

where `\(N(a, b; t_1, t_2)\)` is a normally distributed random variable with mean a and variance b. The parameters t,,1,, and t,,2,, make explicit the statistical independence of *N* on different time intervals; that is, if `\([t_1, t_2)\)` and `\([t_3, t_4)\)` are disjoint intervals, then `\(N(a, b; t_1, t_2)\)` and `\(N(a, b; t_3, t_4)\)` are independent.

The calculation is actually very simple. A naive implementation that prints *n* steps of the Brownian motion might look like this:

```
from scipy.stats import norm

# Process parameters
delta = 0.25
dt = 0.1

# Initial condition.
x = 0.0

# Number of iterations to compute.
n = 20

# Iterate to compute the steps of the Brownian motion.
for k in range(n):
    x = x + norm.rvs(scale=delta**2*dt)
    print x
```

```
0.0149783802189
0.0153383445186
0.0234318982959
0.0212808305024
0.0114258184942
0.01354252966
0.024915691657
0.0225717389674
0.0202001899576
0.0210395736257
0.0346234119557
0.0315723937897
0.0296012566384
0.0296135943506
0.0198499273499
0.0165205162055
0.00688369775327
0.0069949507719
0.00888200058681
0.00297662267152
```

The above code could be easily modified to save the iterations in an array instead of printing them.

The problem with the above code is that it is slow. If we want to compute a large number of iterations, we can do much better. The key is to note that the calculation is the cumulative sum of samples from the normal distribution. A fast version can be implemented by first generating all the samples from the normal distribution with one call to scipy.stats.norm.rvs(), and then using the numpy *cumsum* function to form the cumulative sum.

The following function uses this idea to implement the function *brownian()*. The function allows the initial condition to be an array (or anything that can be converted to an array). Each element of *x0* is treated as an initial condition for a Brownian motion.

```
"""
brownian() implements one dimensional Brownian motion (i.e. the Wie
"""

# File: brownian.py

from math import sqrt
from scipy.stats import norm
import numpy as np

def brownian(x0, n, dt, delta, out=None):
    """
 Generate an instance of Brownian motion (i.e. the Wiener process)
```

```
X(t) = X(0) + N(0, delta**2 * t; 0, t)

where N(a,b; t0, t1) is a normally distributed random variable wit
variance b.  The parameters t0 and t1 make explicit the statistica
independence of N on different time intervals; that is, if [t0, t1
[t2, t3) are disjoint intervals, then N(a, b; t0, t1) and N(a, b;
are independent.

Written as an iteration scheme,

X(t + dt) = X(t) + N(0, delta**2 * dt; t, t+dt)

If `x0` is an array (or array-like), each value in `x0` is treatec
an initial condition, and the value returned is a numpy array with
more dimension than `x0`.

Arguments
---------
x0 : float or numpy array (or something that can be converted to a
using numpy.asarray(x0)).
The initial condition(s) (i.e. position(s)) of the Brownian motior
n : int
The number of steps to take.
dt : float
The time step.
delta : float
delta determines the "speed" of the Brownian motion.  The random v
of the position at time t, X(t), has a normal distribution whose m
the position at time t=0 and whose variance is delta**2*t.
out : numpy array or None
If `out` is not None, it specifies the array in which to put the
result.  If `out` is None, a new numpy array is created and returr

Returns
-------
A numpy array of floats with shape `x0.shape + (n,)`.

Note that the initial value `x0` is not included in the returned a
"""

   x0 = np.asarray(x0)

   # For each element of x0, generate a sample of n numbers from a
   # normal distribution.
   r = norm.rvs(size=x0.shape + (n,), scale=delta*sqrt(dt))

   # If `out` was not given, create an output array.
   if out is None:
       out = np.empty(r.shape)

   # This computes the Brownian motion by forming the cumulative s
   # the random samples.
   np.cumsum(r, axis=-1, out=out)
```

```
    # Add the initial condition.
    out += np.expand_dims(x0, axis=-1)

    return out
```

Example

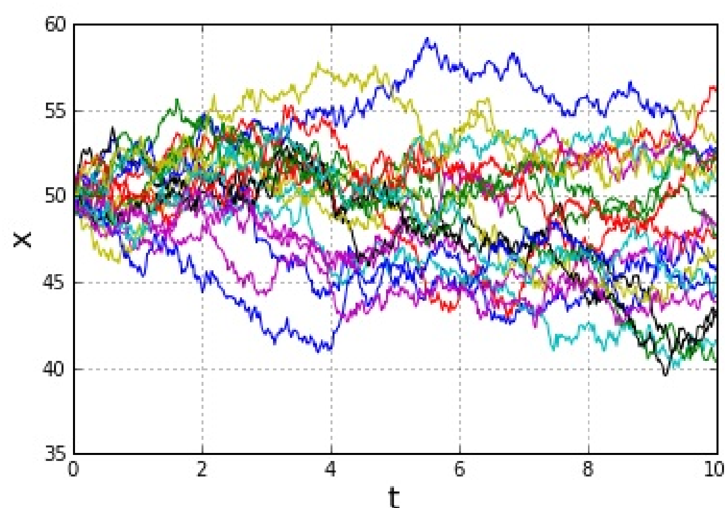
Here's a script that uses this function and matplotlib's pylab module to plot several realizations of Brownian motion.

```
import numpy
from pylab import plot, show, grid, xlabel, ylabel

# The Wiener process parameter.
delta = 2
# Total time.
T = 10.0
# Number of steps.
N = 500
# Time step size
dt = T/N
# Number of realizations to generate.
m = 20
# Create an empty array to store the realizations.
x = numpy.empty((m,N+1))
# Initial values of x.
x[:, 0] = 50

brownian(x[:,0], N, dt, delta, out=x[:,1:])

t = numpy.linspace(0.0, N*dt, N+1)
for k in range(m):
    plot(t, x[k])
xlabel('t', fontsize=16)
ylabel('x', fontsize=16)
grid(True)
show()
```



2D Brownian Motion

The same function can be used to generate Brownian motion in two dimensions, since each dimension is just a one-dimensional Brownian motion.

The following script provides a demo.

```
import numpy
from pylab import plot, show, grid, axis, xlabel, ylabel, title

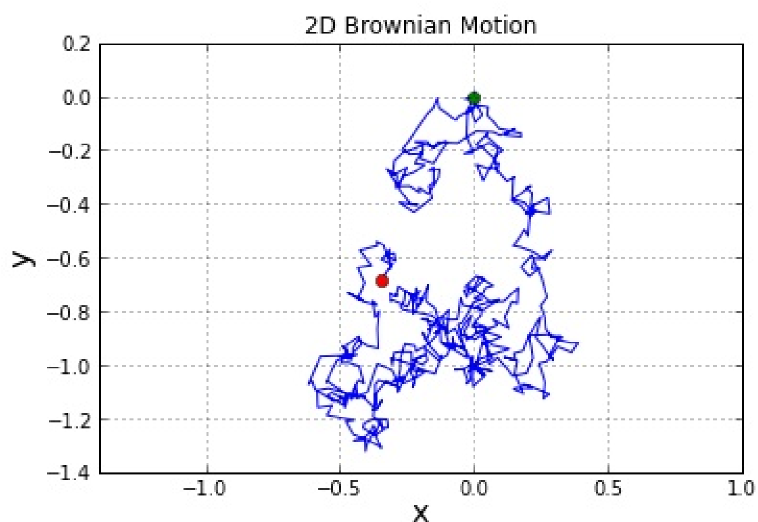
# The Wiener process parameter.
delta = 0.25
# Total time.
T = 10.0
# Number of steps.
N = 500
# Time step size
dt = T/N
# Initial values of x.
x = numpy.empty((2,N+1))
x[:, 0] = 0.0

brownian(x[:,0], N, dt, delta, out=x[:,1:])

# Plot the 2D trajectory.
plot(x[0],x[1])

# Mark the start and end points.
plot(x[0,0],x[1,0], 'go')
plot(x[0,-1], x[1,-1], 'ro')

# More plot decorations.
title('2D Brownian Motion')
xlabel('x', fontsize=16)
ylabel('y', fontsize=16)
axis('equal')
grid(True)
show()
```



Butterworth Bandpass

This cookbook recipe demonstrates the use of `scipy.signal.butter` to create a bandpass Butterworth filter. `scipy.signal.freqz` is used to compute the frequency response, and `scipy.signal.lfilter` is used to apply the filter to a signal. (This code was originally given in an answer to a question at stackoverflow.com.)

```
from scipy.signal import butter, lfilter

def butter_bandpass(lowcut, highcut, fs, order=5):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    b, a = butter(order, [low, high], btype='band')
    return b, a

def butter_bandpass_filter(data, lowcut, highcut, fs, order=5):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    y = lfilter(b, a, data)
    return y

def run():
    import numpy as np
    import matplotlib.pyplot as plt
    from scipy.signal import freqz

    # Sample rate and desired cutoff frequencies (in Hz).
    fs = 5000.0
    lowcut = 500.0
    highcut = 1250.0

    # Plot the frequency response for a few different orders.
    plt.figure(1)
    plt.clf()
    for order in [3, 6, 9]:
        b, a = butter_bandpass(lowcut, highcut, fs, order=order)
        w, h = freqz(b, a, worN=2000)
        plt.plot((fs * 0.5 / np.pi) * w, abs(h), label="order = %d'

    plt.plot([0, 0.5 * fs], [np.sqrt(0.5), np.sqrt(0.5)],
             '--', label='sqrt(0.5)')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Gain')
    plt.grid(True)
    plt.legend(loc='best')

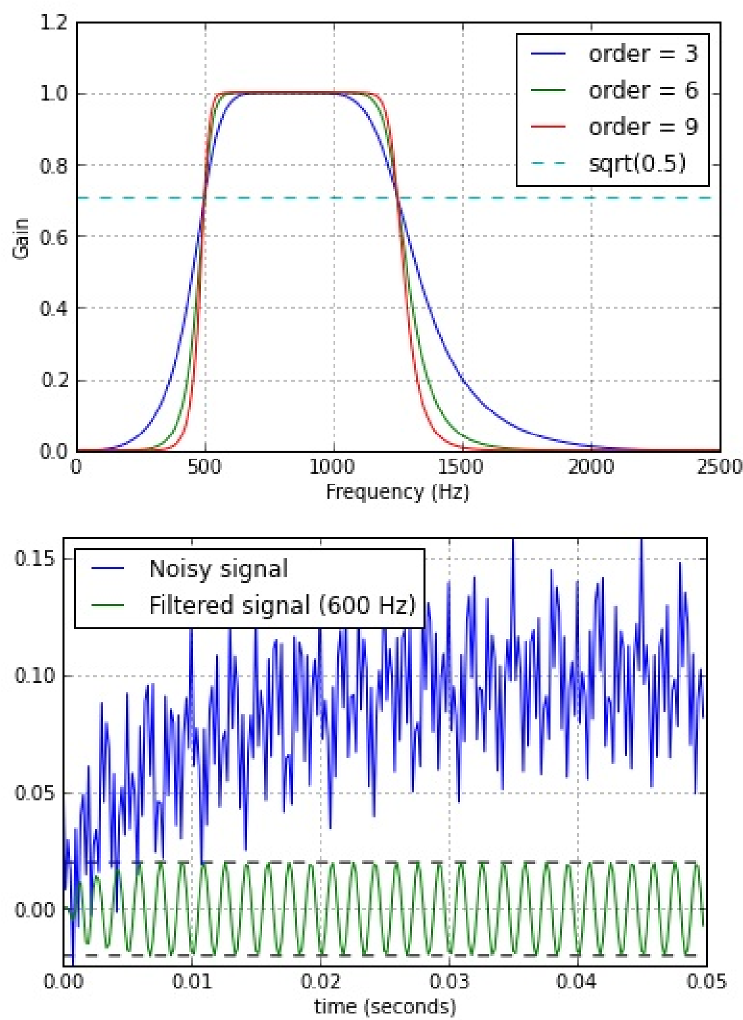
    # Filter a noisy signal.
    T = 0.05
```

```
    nsamples = T * fs
    t = np.linspace(0, T, nsamples, endpoint=False)
    a = 0.02
    f0 = 600.0
    x = 0.1 * np.sin(2 * np.pi * 1.2 * np.sqrt(t))
    x += 0.01 * np.cos(2 * np.pi * 312 * t + 0.1)
    x += a * np.cos(2 * np.pi * f0 * t + .11)
    x += 0.03 * np.cos(2 * np.pi * 2000 * t)
    plt.figure(2)
    plt.clf()
    plt.plot(t, x, label='Noisy signal')

    y = butter_bandpass_filter(x, lowcut, highcut, fs, order=6)
    plt.plot(t, y, label='Filtered signal (%g Hz)' % f0)
    plt.xlabel('time (seconds)')
    plt.hlines([-a, a], 0, T, linestyles='--')
    plt.grid(True)
    plt.axis('tight')
    plt.legend(loc='upper left')

    plt.show()

run()
```



order = 3
order = 6
order = 9
sqrt(0.5)
Gain
Frequency (Hz)
Noisy signal
Filtered signal (600 Hz)
time (seconds)

Communication theory

These two examples illustrate simple simulation of a digital BPSK modulated communication system where only one sample per symbol is used, and signal is affected only by AWGN noise.

In the first example, we cycle through different signal to noise values, and the signal length is a function of theoretical probability of error. As a rule of thumb, we want to count about 100 errors for each SNR value, which determines the length of the signal (and noise) vector(s).

```
#!/usr/bin/python
# BPSK digital modulation example
# by Ivo Maljevic

from numpy import *
from scipy.special import erfc
import matplotlib.pyplot as plt

SNR_MIN     = 0
SNR_MAX     = 9
Eb_No_dB    = arange(SNR_MIN,SNR_MAX+1)
SNR         = 10**(Eb_No_dB/10.0)  # linear SNR

Pe          = empty(shape(SNR))
BER         = empty(shape(SNR))

loop = 0
for snr in SNR:      # SNR loop
 Pe[loop] = 0.5*erfc(sqrt(snr))
 VEC_SIZE = ceil(100/Pe[loop])  # vector length is a function of Pe

 # signal vector, new vector for each SNR value
 s = 2*random.randint(0,high=2,size=VEC_SIZE)-1

 # linear power of the noise; average signal power = 1
 No = 1.0/snr

 # noise
 n = sqrt(No/2)*random.randn(VEC_SIZE)

 # signal + noise
 x = s + n

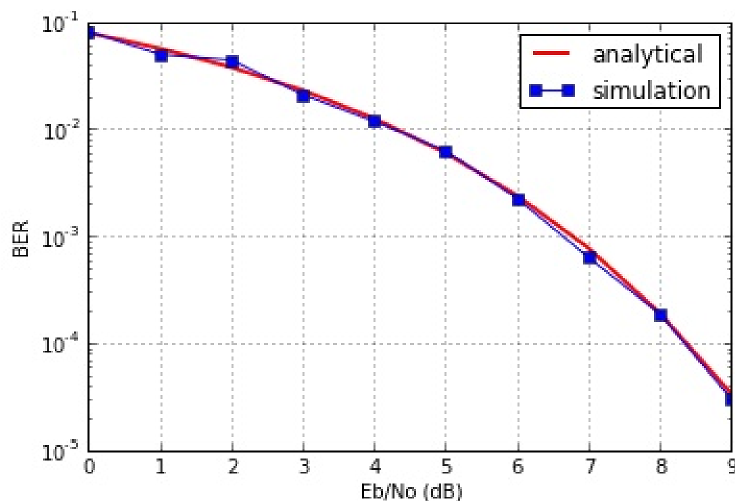
 # decode received signal + noise
 y = sign(x)

 # find erroneous symbols
 err = where(y != s)
```

```
    error_sum = float(len(err[0]))
    BER[loop] = error_sum/VEC_SIZE
    print 'Eb_No_dB=%4.2f, BER=%10.4e, Pe=%10.4e' % \
           (Eb_No_dB[loop], BER[loop], Pe[loop])
    loop += 1

#plt.semilogy(Eb_No_dB, Pe,'r',Eb_No_dB, BER,'s')
plt.semilogy(Eb_No_dB, Pe,'r',linewidth=2)
plt.semilogy(Eb_No_dB, BER,'-s')
plt.grid(True)
plt.legend(('analytical','simulation'))
plt.xlabel('Eb/No (dB)')
plt.ylabel('BER')
plt.show()
```

```
Eb_No_dB=0.00, BER=8.0975e-02, Pe=7.8650e-02
Eb_No_dB=1.00, BER=4.9522e-02, Pe=5.6282e-02
Eb_No_dB=2.00, BER=4.3870e-02, Pe=3.7506e-02
Eb_No_dB=3.00, BER=2.0819e-02, Pe=2.2878e-02
Eb_No_dB=4.00, BER=1.1750e-02, Pe=1.2501e-02
Eb_No_dB=5.00, BER=6.2515e-03, Pe=5.9539e-03
Eb_No_dB=6.00, BER=2.2450e-03, Pe=2.3883e-03
Eb_No_dB=7.00, BER=6.3359e-04, Pe=7.7267e-04
Eb_No_dB=8.00, BER=1.8709e-04, Pe=1.9091e-04
Eb_No_dB=9.00, BER=3.0265e-05, Pe=3.3627e-05
```



In the second, slightly modified example, the problem of signal length growth is solved by braking a signal into frames.Namely, the number of samples for a given SNR grows quickly, so that the simulation above is not practical for Eb/No values greater than 9 or 10 dB.

```
#!/usr/bin/python
# BPSK digital modulation: modified example
```

```
# by Ivo Maljevic

from scipy import *
from math import sqrt, ceil  # scalar calls are faster
from scipy.special import erfc
import matplotlib.pyplot as plt

rand   = random.rand
normal = random.normal

SNR_MIN   = 0
SNR_MAX   = 10
FrameSize = 10000
Eb_No_dB  = arange(SNR_MIN,SNR_MAX+1)
Eb_No_lin = 10**(Eb_No_dB/10.0)  # linear SNR

# Allocate memory
Pe        = empty(shape(Eb_No_lin))
BER       = empty(shape(Eb_No_lin))

# signal vector (for faster exec we can repeat the same frame)
s = 2*random.randint(0,high=2,size=FrameSize)-1

loop = 0
for snr in Eb_No_lin:
 No        = 1.0/snr
 Pe[loop]  = 0.5*erfc(sqrt(snr))
 nFrames   = ceil(100.0/FrameSize/Pe[loop])
 error_sum = 0
 scale = sqrt(No/2)

 for frame in arange(nFrames):
   # noise
   n = normal(scale=scale, size=FrameSize)

   # received signal + noise
   x = s + n

   # detection (information is encoded in signal phase)
   y = sign(x)

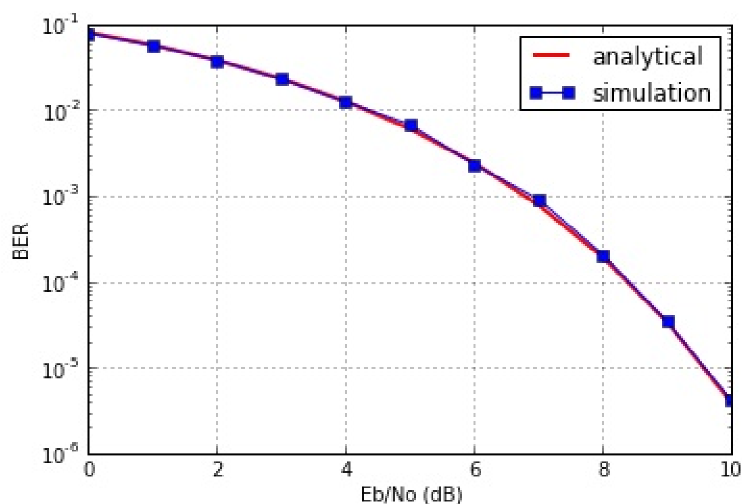
   # error counting
   err = where (y != s)
   error_sum += len(err[0])

   # end of frame loop
   ##################################################

 BER[loop] = error_sum/(FrameSize*nFrames)  # SNR loop level
 print 'Eb_No_dB=%2d, BER=%10.4e, Pe[loop]=%10.4e' % \
        (Eb_No_dB[loop], BER[loop], Pe[loop])
 loop += 1
```

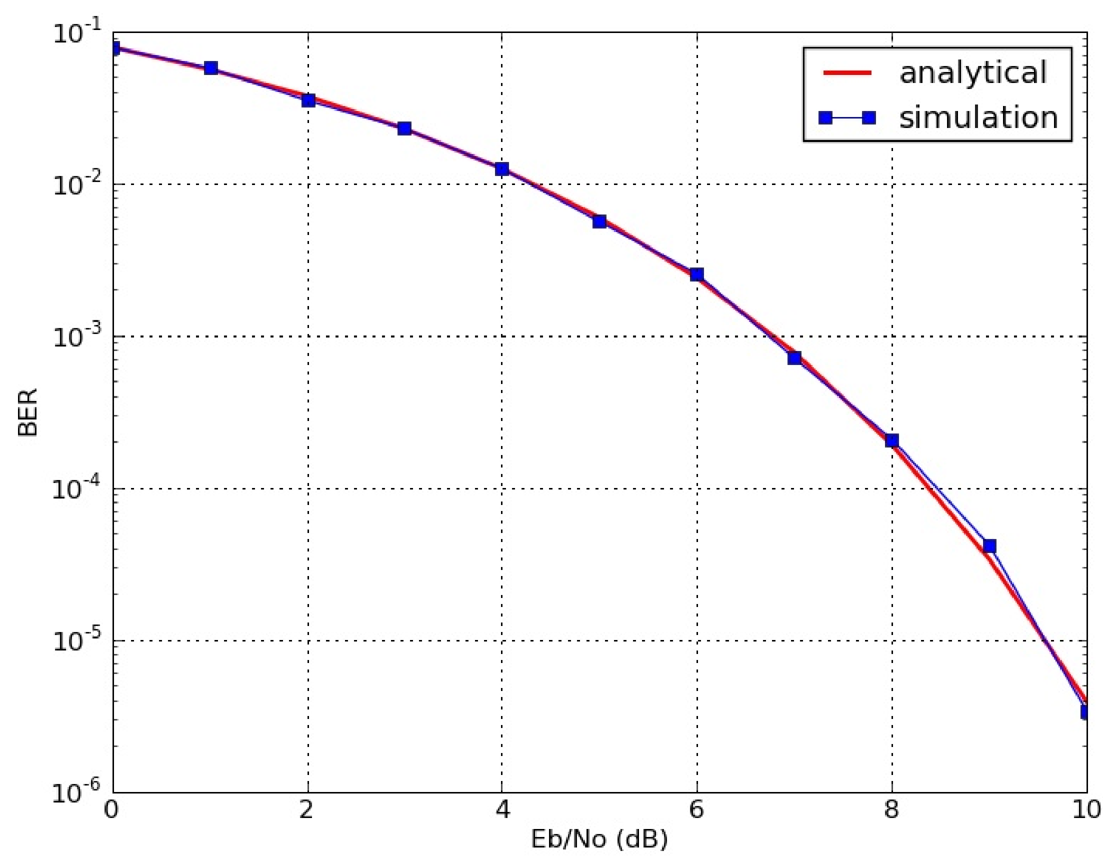
```
plt.semilogy(Eb_No_dB, Pe,'r',linewidth=2)
plt.semilogy(Eb_No_dB, BER,'-s')
plt.grid(True)
plt.legend(('analytical','simulation'))
plt.xlabel('Eb/No (dB)')
plt.ylabel('BER')
plt.show()
```

```
Eb_No_dB= 0, BER=7.6900e-02, Pe[loop]=7.8650e-02
Eb_No_dB= 1, BER=5.5800e-02, Pe[loop]=5.6282e-02
Eb_No_dB= 2, BER=3.7600e-02, Pe[loop]=3.7506e-02
Eb_No_dB= 3, BER=2.2600e-02, Pe[loop]=2.2878e-02
Eb_No_dB= 4, BER=1.2300e-02, Pe[loop]=1.2501e-02
Eb_No_dB= 5, BER=6.6500e-03, Pe[loop]=5.9539e-03
Eb_No_dB= 6, BER=2.3000e-03, Pe[loop]=2.3883e-03
Eb_No_dB= 7, BER=9.0000e-04, Pe[loop]=7.7267e-04
Eb_No_dB= 8, BER=2.0566e-04, Pe[loop]=1.9091e-04
Eb_No_dB= 9, BER=3.3893e-05, Pe[loop]=3.3627e-05
Eb_No_dB=10, BER=4.1425e-06, Pe[loop]=3.8721e-06
```



Attachments

- `BPSK_BER.PNG`



analytical
simulation
BER
Eb/No (dB)
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
0
2
4
6
8
10

Correlated Random Samples

Note: *This cookbook entry shows how to generate random samples from a multivariate normal distribution using tools from SciPy, but in fact NumPy includes the function* `numpy.random.multivariate_normal` *to accomplish the same task.*

To generate correlated normally distributed random samples, one can first generate uncorrelated samples, and then multiply them by a matrix *C* such that `\(C C^T = R\)` , where *R* is the desired covariance matrix. *C* can be created, for example, by using the Cholesky decomposition of *R*, or from the eigenvalues and eigenvectors of *R*.

```
"""Example of generating correlated normally distributed random sar

import numpy as np
from scipy.linalg import eigh, cholesky
from scipy.stats import norm

from pylab import plot, show, axis, subplot, xlabel, ylabel, grid

# Choice of cholesky or eigenvector method.
method = 'cholesky'
#method = 'eigenvectors'

num_samples = 400

# The desired covariance matrix.
r = np.array([
        [  3.40, -2.75, -2.00],
        [ -2.75,  5.50,  1.50],
        [ -2.00,  1.50,  1.25]
    ])

# Generate samples from three independent normally distributed ranc
# variables (with mean 0 and std. dev. 1).
x = norm.rvs(size=(3, num_samples))

# We need a matrix `c` for which `c*c^T = r`.  We can use, for exar
# the Cholesky decomposition, or the we can construct `c` from the
# eigenvectors and eigenvalues.

if method == 'cholesky':
    # Compute the Cholesky decomposition.
    c = cholesky(r, lower=True)
else:
    # Compute the eigenvalues and eigenvectors.
    evals, evecs = eigh(r)
    # Construct c, so c*c^T = r.
```

```
    c = np.dot(evecs, np.diag(np.sqrt(evals)))

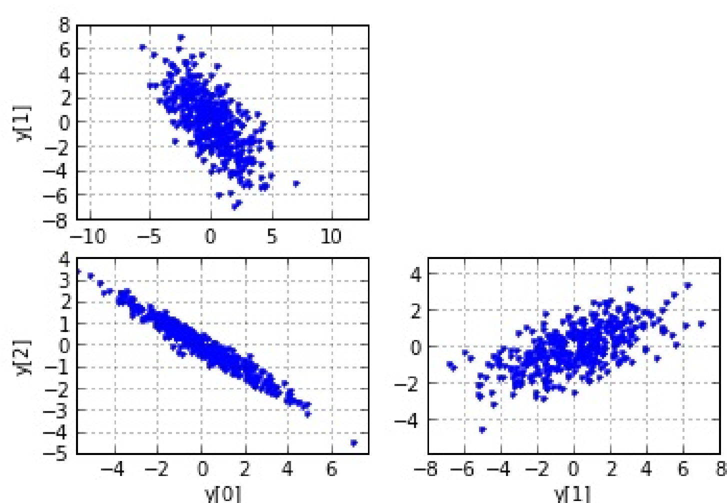
# Convert the data to correlated random variables.
y = np.dot(c, x)

#
# Plot various projections of the samples.
#
subplot(2,2,1)
plot(y[0], y[1], 'b.')
ylabel('y[1]')
axis('equal')
grid(True)

subplot(2,2,3)
plot(y[0], y[2], 'b.')
xlabel('y[0]')
ylabel('y[2]')
axis('equal')
grid(True)

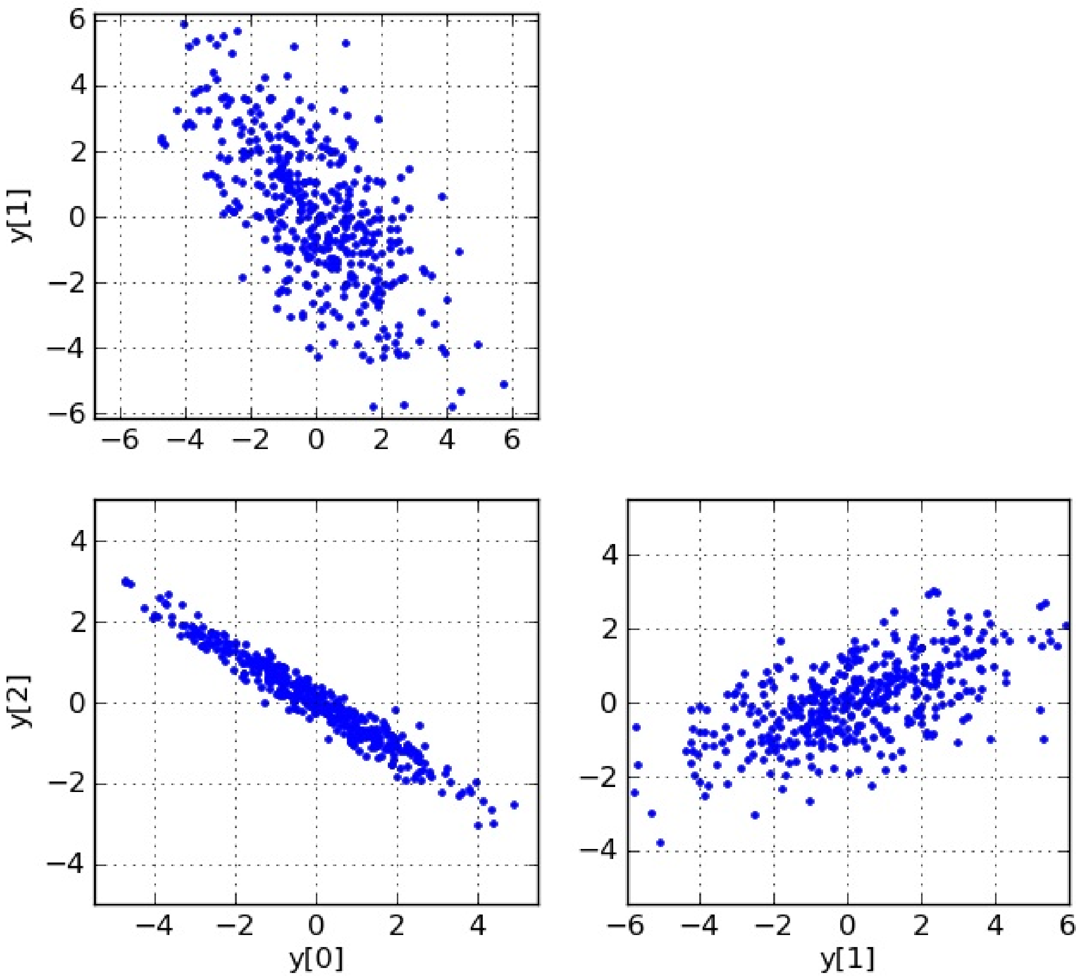
subplot(2,2,4)
plot(y[1], y[2], 'b.')
xlabel('y[1]')
axis('equal')
grid(True)

show()
```



Attachments

- `correlated_random_vars.png`


y[1]
y[2]
y[0]
y[1]

Easy multithreading

Python includes a multithreading package, “threading”, but python’s multithreading is seriously limited by the Global Interpreter Lock, which allows only one thread to be interacting with the interpreter at a time. For purely interpreted code, this makes multithreading effectively cooperative and unable to take advantage of multiple cores.

However, numpy code often releases the GIL while it is calculating, so that simple parallelism can speed up the code. For sophisticated applications, one should look into MPI or using threading directly, but surprisingly often one’s application is “embarrassingly parallel”, that is, one simply has to do the same operation to many objects, with no interaction between iterations. This kind of calculation can be easily parallelized:

```
dft = parallel_map(lambda f: sum(exp(2.j*pi*f*times)), frequencies
```

The code implementing parallel_map is not too complicated, and is attached to this entry. Even simpler, if one doesn’t want to return values:

```
def compute(n):
    ...do something...
foreach(compute, range(100))
```

This replaces a for loop.

See attachments for code (written by AMArchibald). [[AttachList]]

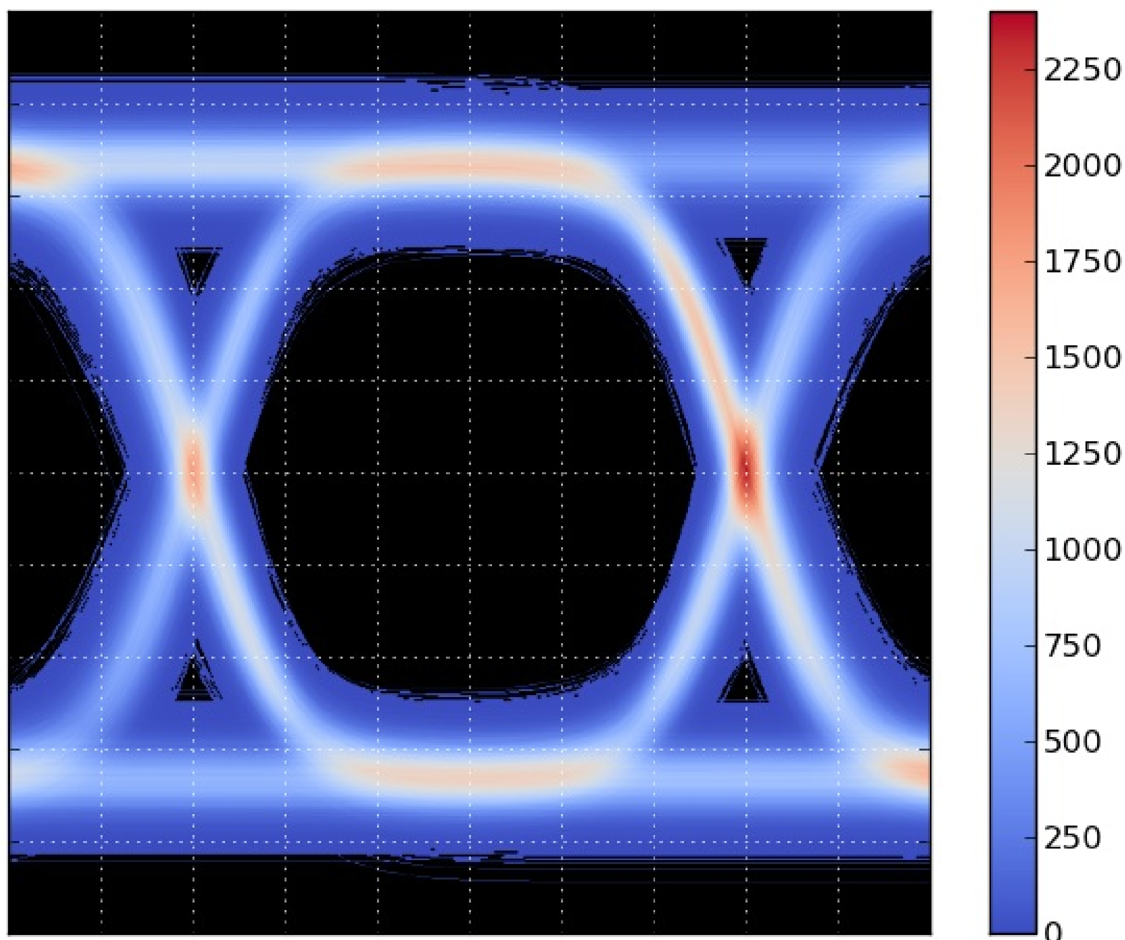
See also ParallelProgramming for alternatives and more discussion.

Attachments

- `handythread.py`
- `test_handythread.py`

Eye Diagram

The code below generates the following plot:



The main script generates `num_traces` traces, and on a grid of 600x600, it counts the number times a trace crosses a grid point. The grid is then plotted using matplotlib's imshow() function. The counting is performed using Bresenham's line algorithm, to ensure that the counting is correct, and steep parts of the curve don't result in missed counts.

Bresenham's algorithm is slow in pure Python, so a Cython version is included. If you do not build the Cython version of the Bresenham code, be sure to reduce `num_traces` before running the program!

Here's the main demo script, eye_demo.py.

```
#!python
import numpy as np

use_fast = True
try:
    from brescount import bres_curve_count
```

```
except ImportError:
    print "The cython version of the curve counter is not available
    use_fast = False

def bres_segment_count_slow(x0, y0, x1, y1, grid):
    """Bresenham's algorithm.

 The value of grid[x,y] is incremented for each x,y
 in the line from (x0,y0) up to but not including (x1, y1).
 """

    nrows, ncols = grid.shape

    dx = abs(x1 - x0)
    dy = abs(y1 - y0)

    sx = 0
    if x0 < x1:
        sx = 1
    else:
        sx = -1
    sy = 0
    if y0 < y1:
        sy = 1
    else:
        sy = -1

    err = dx - dy

    while True:
        # Note: this test is moved before setting
        # the value, so we don't set the last point.
        if x0 == x1 and y0 == y1:
            break

        if 0 <= x0 < nrows and 0 <= y0 < ncols:
            grid[x0, y0] += 1

        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x0 += sx
        if e2 < dx:
            err += dx
            y0 += sy

def bres_curve_count_slow(x, y, grid):
    for k in range(x.size - 1):
        x0 = x[k]
        y0 = y[k]
        x1 = x[k+1]
        y1 = y[k+1]
        bres_segment_count_slow(x0, y0, x1, y1, grid)
```

```
def random_trace(t):
    s = 2*(np.random.randint(0, 5) % 2) - 1
    r = 0.01 * np.random.randn()
    s += r
    a = 2.0 + 0.001 * np.random.randn()
    q = 2*(np.random.randint(0, 7) % 2) - 1
    t2 = t + q*(6 + 0.01*np.random.randn())
    t2 += 0.05*np.random.randn()*t
    y = a * (np.exp(s*t2) / (1 + np.exp(s*t2)) - 0.5) + 0.07*np.ran
    return y

if __name__ == "__main__":
    import matplotlib.pyplot as plt
    grid_size = 600
    grid = np.zeros((grid_size, grid_size), dtype=np.int32)

    tmin = -10.0
    tmax = 10.0
    n = 81
    t = np.linspace(tmin, tmax, n)
    dt = (tmax - tmin) / (n - 1)

    ymin = -1.5
    ymax = 1.5

    num_traces = 1000

    for k in range(num_traces):

        # Add some noise to the times at which the signal
        # will be sampled.  Without this, all the samples occur
        # at the same times, and this produces an aliasing
        # effect in the resulting bin counts.
        # If n == grid_size, this can be dropped, and t2 = t
        # can be used instead. (Or, implement an antialiased
        # version of bres_curve_count.)
        steps = dt + np.sqrt(0.01 * dt) * np.random.randn(n)
        steps[0] = 0
        steps_sum = steps.cumsum()
        t2 = tmin + (tmax - tmin) * steps_sum / steps_sum[-1]

        td = (((t2 - tmin) / (tmax - tmin)) * grid_size).astype(np.

        y = random_trace(t2)

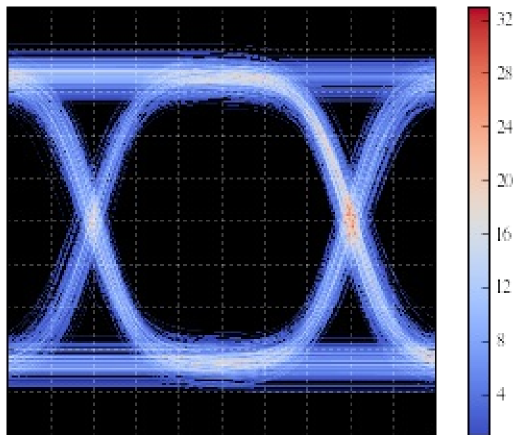
        # Convert y to integers in the range [0,grid_size).
        yd = (((y - ymin) / (ymax - ymin)) * grid_size).astype(np.i

        if use_fast:
            bres_curve_count(td, yd, grid)
        else:
            bres_curve_count_slow(td, yd, grid)
```

```
    plt.figure()
    # Convert to float32 so we can use nan instead of 0.
    grid = grid.astype(np.float32)
    grid[grid==0] = np.nan
    plt.grid(color='w')
    plt.imshow(grid.T[::-1,:], extent=[0,1,0,1], cmap=plt.cm.coolwa
               interpolation='gaussian')
    ax = plt.gca()
    ax.set_axis_bgcolor('k')
    ax.set_xticks(np.linspace(0,1,11))
    ax.set_yticks(np.linspace(0,1,11))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    plt.colorbar()
    fig = plt.gcf()

    #plt.savefig("eye-diagram.png", bbox_inches='tight')
    plt.show()
```

```
The cython version of the curve counter is not available.
```



Here's brescount.pyx, the Cython implementation of Bresenham's line algorithm:

```
#!python
import numpy as np
cimport numpy as np
cimport cython

@cython.boundscheck(False)
cdef int bres_segment_count(unsigned x0, unsigned y0,
                            unsigned x1, unsigned y1,
                            np.ndarray[np.int32_t, ndim=2] grid):
    """Bresenham's algorithm.
```

```
    See http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm
    """

    cdef unsigned nrows, ncols
    cdef int e2, sx, sy, err
    cdef int dx, dy

    nrows = grid.shape[0]
    ncols = grid.shape[1]

    if x1 > x0:
        dx = x1 - x0
    else:
        dx = x0 - x1
    if y1 > y0:
        dy = y1 - y0
    else:
        dy = y0 - y1

    sx = 0
    if x0 < x1:
        sx = 1
    else:
        sx = -1
    sy = 0
    if y0 < y1:
        sy = 1
    else:
        sy = -1

    err = dx - dy

    while True:
        # Note: this test occurs before increment the
        # grid value, so we don't count the last point.
        if x0 == x1 and y0 == y1:
            break

        if (x0 < nrows) and (y0 < ncols):
            grid[x0, y0] += 1

        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x0 += sx
        if e2 < dx:
            err += dx
            y0 += sy

    return 0

def bres_curve_count(np.ndarray[np.int32_t, ndim=1] x,
                     np.ndarray[np.int32_t, ndim=1] y,
```

```
                      np.ndarray[np.int32_t, ndim=2] grid):
    cdef unsigned k
    cdef int x0, y0, x1, y1

    for k in range(len(x)-1):
        x0 = x[k]
        y0 = y[k]
        x1 = x[k+1]
        y1 = y[k+1]
        bres_segment_count(x0, y0, x1, y1, grid)
    if 0 <= x1 < grid.shape[0] and 0 <= y1 < grid.shape[1]:
        grid[x1, y1] += 1
```

This file, setup.py, can be used to build the Cython extension module:

```
#!python
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

import numpy

ext = Extension("brescount", ["brescount.pyx"],
    include_dirs = [numpy.get_include()])

setup(ext_modules=[ext],
      cmdclass = {'build_ext': build_ext})
```

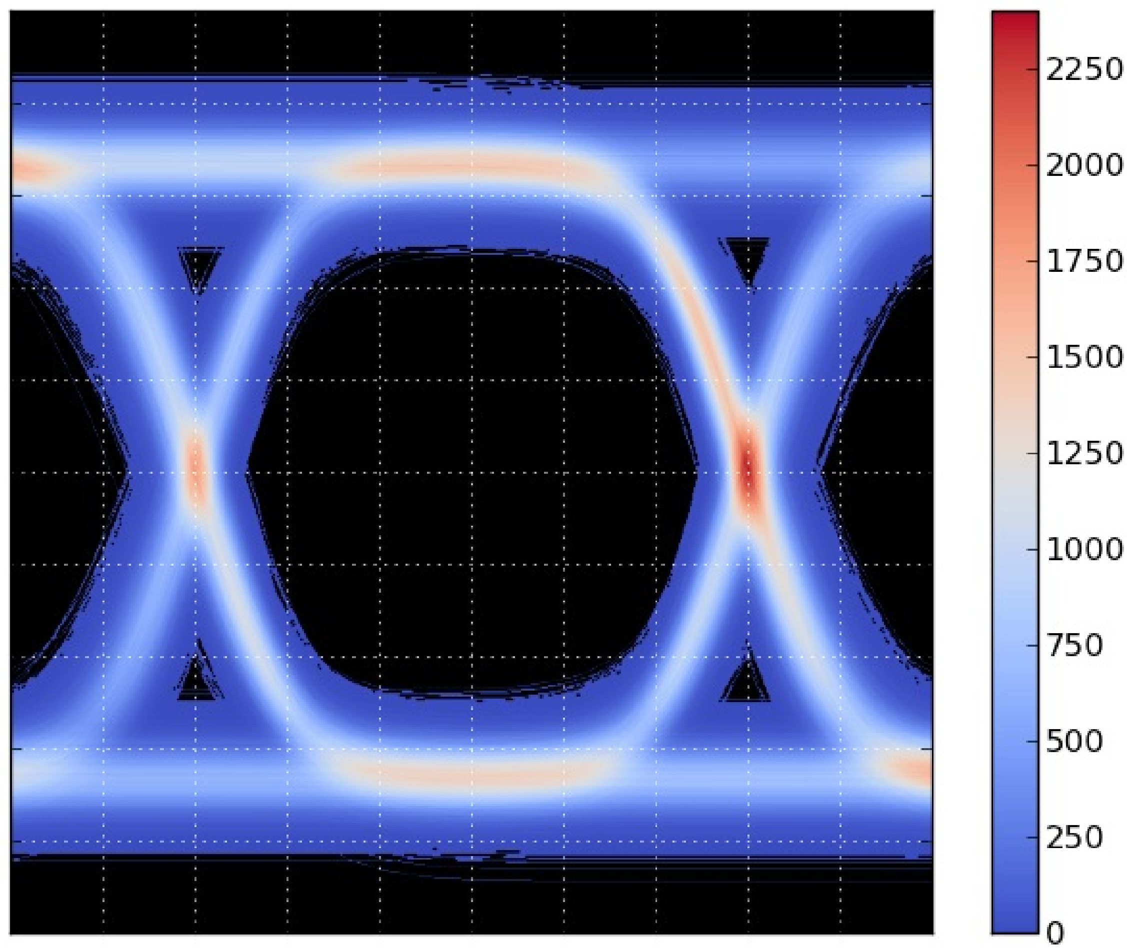
To build the extension module, you must have Cython installed.

You can build the extension module as follows:

```
$ python setup.py build_ext --inplace
```

Attachments

- `eye-diagram3.png`


2250
2000
1750
1500
1250
1000
750
500
250
0

FIR filter

This cookbook example shows how to design and use a low-pass FIR filter using functions from scipy.signal.

The pylab module from matplotlib is used to create plots.

```
#!python

from numpy import cos, sin, pi, absolute, arange
from scipy.signal import kaiserord, lfilter, firwin, freqz
from pylab import figure, clf, plot, xlabel, ylabel, xlim, ylim, ti

#------------------------------------------------
# Create a signal for demonstration.
#------------------------------------------------

sample_rate = 100.0
nsamples = 400
t = arange(nsamples) / sample_rate
x = cos(2*pi*0.5*t) + 0.2*sin(2*pi*2.5*t+0.1) + \
        0.2*sin(2*pi*15.3*t) + 0.1*sin(2*pi*16.7*t + 0.1) + \
            0.1*sin(2*pi*23.45*t+.8)

#------------------------------------------------
# Create a FIR filter and apply it to x.
#------------------------------------------------

# The Nyquist rate of the signal.
nyq_rate = sample_rate / 2.0

# The desired width of the transition from pass to stop,
# relative to the Nyquist rate.  We'll design the filter
# with a 5 Hz transition width.
width = 5.0/nyq_rate

# The desired attenuation in the stop band, in dB.
ripple_db = 60.0

# Compute the order and Kaiser parameter for the FIR filter.
N, beta = kaiserord(ripple_db, width)

# The cutoff frequency of the filter.
cutoff_hz = 10.0

# Use firwin with a Kaiser window to create a lowpass FIR filter.
taps = firwin(N, cutoff_hz/nyq_rate, window=('kaiser', beta))

# Use lfilter to filter x with the FIR filter.
```

```
filtered_x = lfilter(taps, 1.0, x)

#------------------------------------------------
# Plot the FIR filter coefficients.
#------------------------------------------------

figure(1)
plot(taps, 'bo-', linewidth=2)
title('Filter Coefficients (%d taps)' % N)
grid(True)

#------------------------------------------------
# Plot the magnitude response of the filter.
#------------------------------------------------

figure(2)
clf()
w, h = freqz(taps, worN=8000)
plot((w/pi)*nyq_rate, absolute(h), linewidth=2)
xlabel('Frequency (Hz)')
ylabel('Gain')
title('Frequency Response')
ylim(-0.05, 1.05)
grid(True)

# Upper inset plot.
ax1 = axes([0.42, 0.6, .45, .25])
plot((w/pi)*nyq_rate, absolute(h), linewidth=2)
xlim(0,8.0)
ylim(0.9985, 1.001)
grid(True)

# Lower inset plot
ax2 = axes([0.42, 0.25, .45, .25])
plot((w/pi)*nyq_rate, absolute(h), linewidth=2)
xlim(12.0, 20.0)
ylim(0.0, 0.0025)
grid(True)

#------------------------------------------------
# Plot the original and filtered signals.
#------------------------------------------------

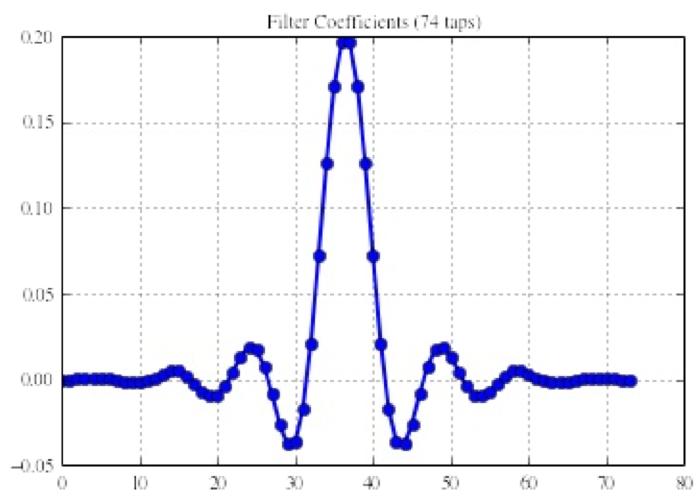
# The phase delay of the filtered signal.
delay = 0.5 * (N-1) / sample_rate

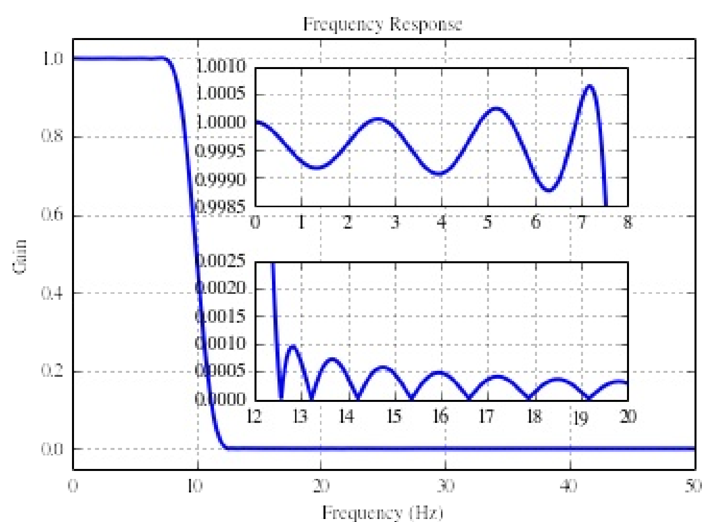
figure(3)
# Plot the original signal.
plot(t, x)
# Plot the filtered signal, shifted to compensate for the phase del
plot(t-delay, filtered_x, 'r-')
# Plot just the "good" part of the filtered signal.  The first N-1
# samples are "corrupted" by the initial conditions.
```

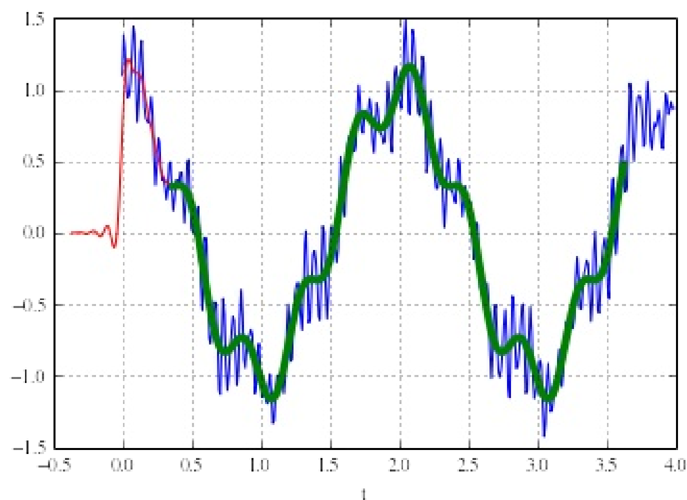
```
plot(t[N-1:]-delay, filtered_x[N-1:], 'g', linewidth=4)

xlabel('t')
grid(True)

show()
```



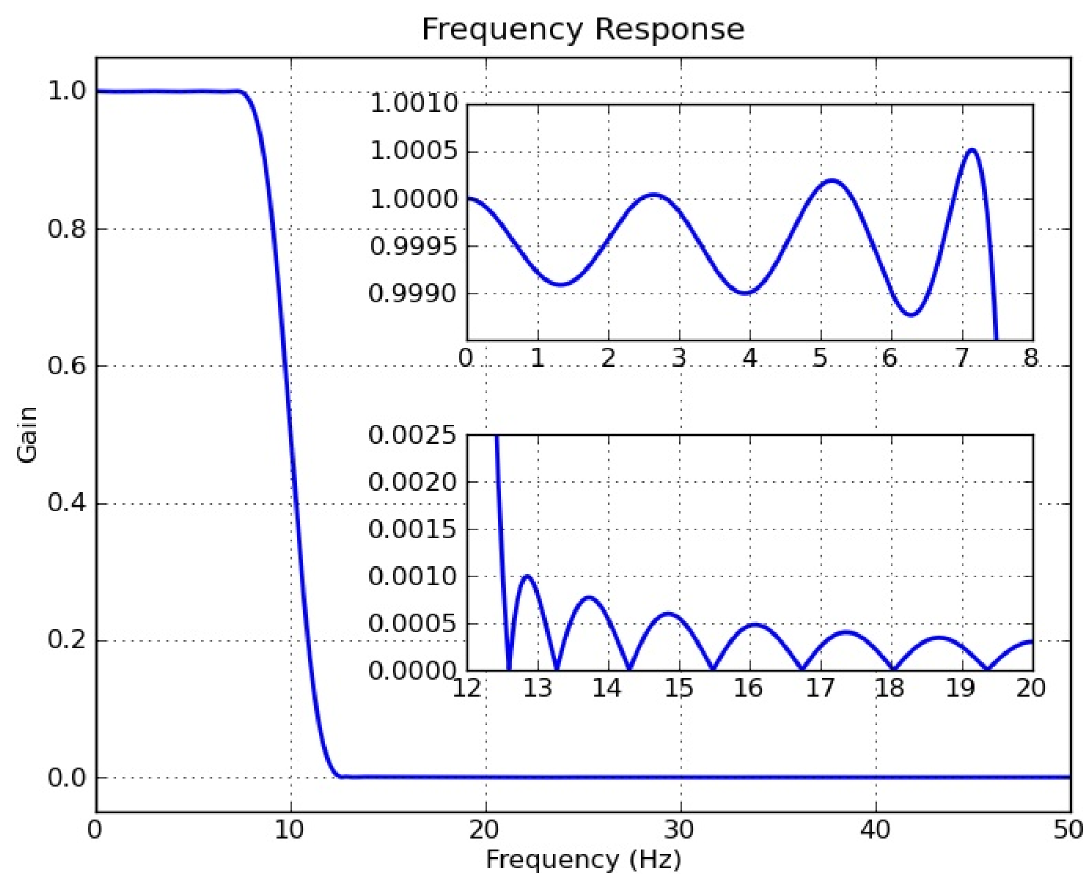


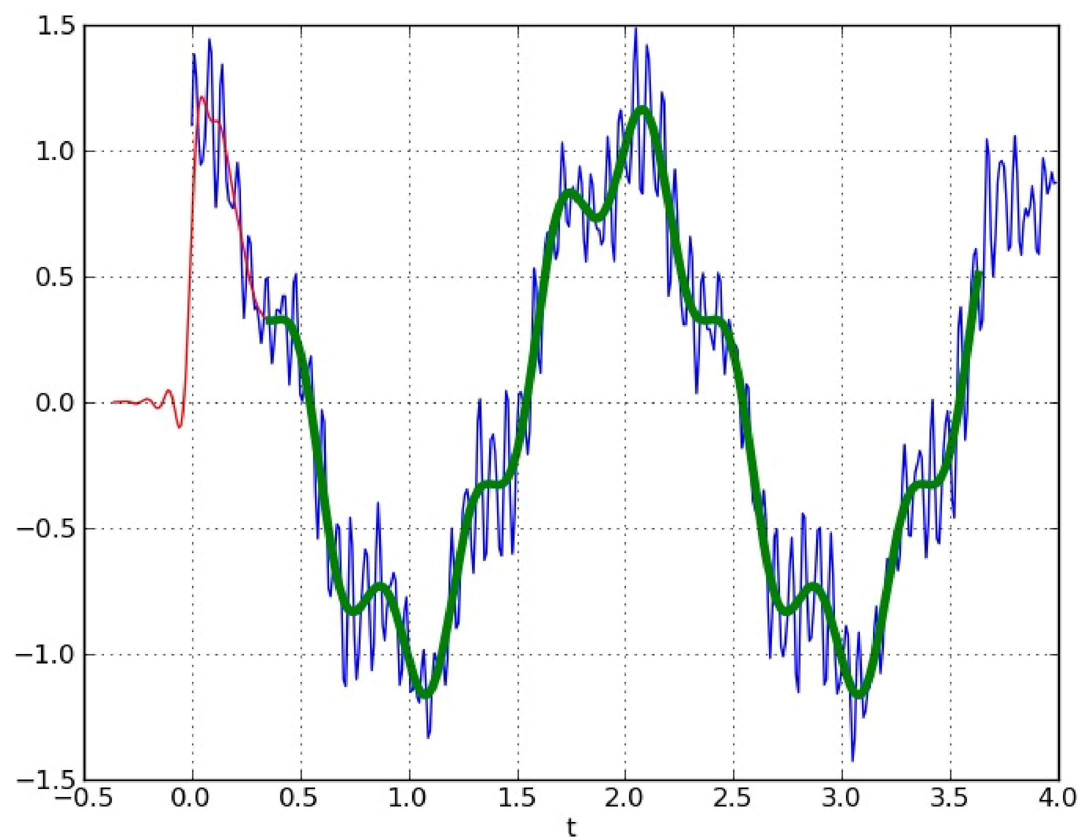


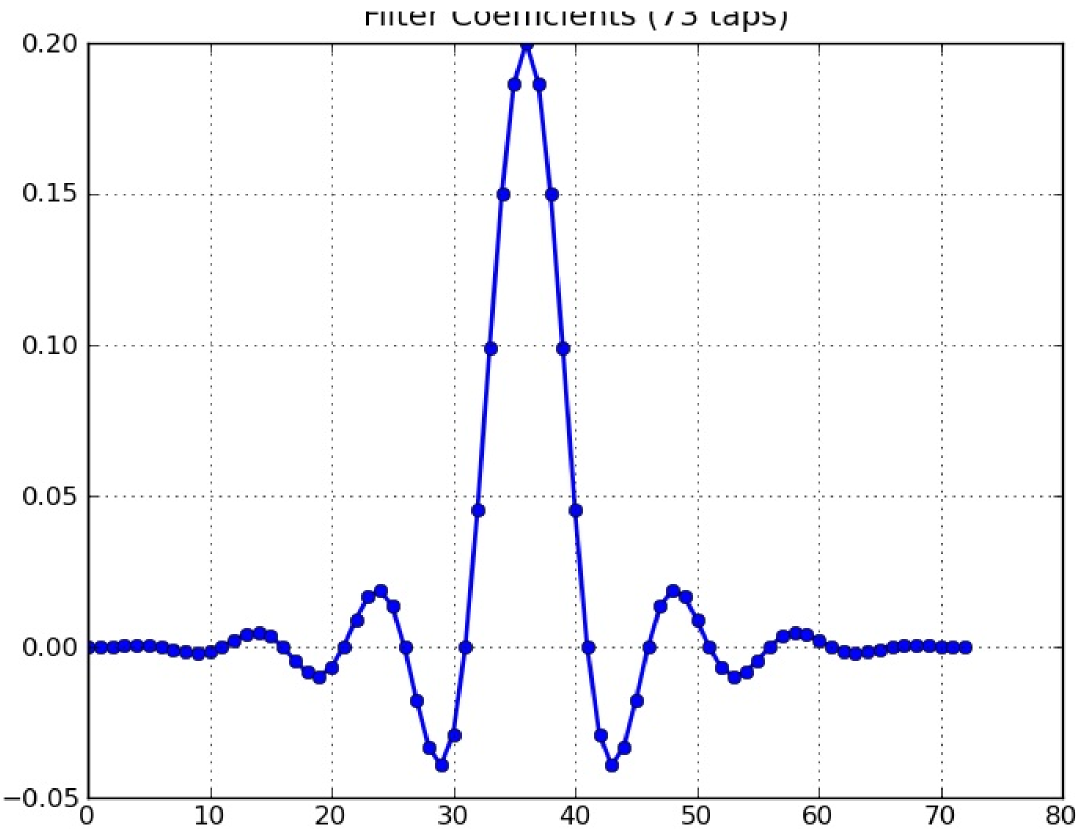
The final plots shows the original signal (thin blue line), the filtered signal (shifted by the appropriate phase delay to align with the original signal; thin red line), and the “good” part of the filtered signal (heavy green line). The “good part” is the part of the signal that is not affected by the initial conditions.

Attachments

- `fir_demo_freq_resp.png`
- `fir_demo_signals.png`
- `fir_demo_taps.png`


Frequency Response
Gain
Frequency (Hz)
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
1.0010
1.0005
1.0000
0.9995
0.9990
0 1 2 3 4 5 6 7 8
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
12 13 14 15 16 17 18 19 20


t
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−0.5 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0


Filter Coefficients (73 taps)
0.20
0.15
0.10
0.05
0.00
−0.05
0
10
20
30
40
50
60
70
80

Filtfilt

This sample code demonstrates the use of the function scipy.signal.filtfilt, a linear filter that achieves zero phase delay by applying an IIR filter to a signal twice, once forwards and once backwards. The order of the filter is twice the original filter order. The function also computes the initial filter parameters in order to provide a more stable response (via lfilter_zi).

For comparison, this script also applies the same IIR filter to the signal using scipy.signal.lfilter; for these calculations, lfilter_zi is used to choose appropriate initial conditions for the filter. Without this, these plots would have long transients near 0. As it is, they have long transients near the initial value of the signal.

Code

```
from numpy import sin, cos, pi, linspace
from numpy.random import randn
from scipy.signal import lfilter, lfilter_zi, filtfilt, butter

from matplotlib.pyplot import plot, legend, show, hold, grid, figu

# Generate a noisy signal to be filtered.
t = linspace(-1, 1, 201)
x = (sin(2 * pi * 0.75 * t*(1-t) + 2.1) + 0.1*sin(2 * pi * 1.25 * t
    0.18*cos(2 * pi * 3.85 * t))
xn = x + randn(len(t)) * 0.08

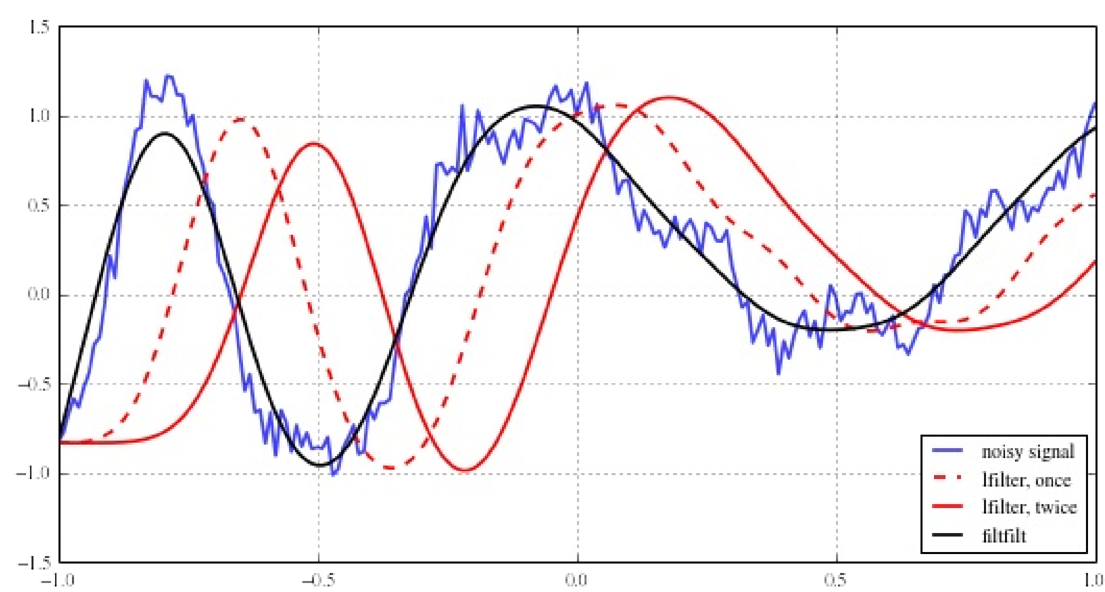
# Create an order 3 lowpass butterworth filter.
b, a = butter(3, 0.05)

# Apply the filter to xn.  Use lfilter_zi to choose the initial co
# of the filter.
zi = lfilter_zi(b, a)
z, _ = lfilter(b, a, xn, zi=zi*xn[0])

# Apply the filter again, to have a result filtered at an order
# the same as filtfilt.
z2, _ = lfilter(b, a, z, zi=zi*z[0])

# Use filtfilt to apply the filter.
y = filtfilt(b, a, xn)

# Make the plot.
figure(figsize=(10,5))
hold(True)
plot(t, xn, 'b', linewidth=1.75, alpha=0.75)
plot(t, z, 'r--', linewidth=1.75)
plot(t, z2, 'r', linewidth=1.75)
plot(t, y, 'k', linewidth=1.75)
legend(('noisy signal',
        'lfilter, once',
        'lfilter, twice',
        'filtfilt'),
        loc='best')
hold(False)
grid(True)
show()
#savefig('plot.png', dpi=65)
```



1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
noisy signal
lfilter, once
lfilter, twice
filtfilt

Finding the Convex Hull of a 2-D Dataset

> **NOTE**: you may want to use use scipy.spatial.ConvexHull instead of this.

This code finds the subsets of points describing the convex hull around a set of 2-D data points. The code optionally uses pylab to animate its progress.

```
import numpy as n, pylab as p, time

def _angle_to_point(point, centre):
    '''calculate angle in 2-D between points and x axis'''
    delta = point - centre
    res = n.arctan(delta[1] / delta[0])
    if delta[0] < 0:
        res += n.pi
    return res

def _draw_triangle(p1, p2, p3, **kwargs):
    tmp = n.vstack((p1,p2,p3))
    x,y = [x[0] for x in zip(tmp.transpose())]
    p.fill(x,y, **kwargs)
    #time.sleep(0.2)

def area_of_triangle(p1, p2, p3):
    '''calculate area of any triangle given co-ordinates of the co
    return n.linalg.norm(n.cross((p2 - p1), (p3 - p1)))/2.

def convex_hull(points, graphic=True, smidgen=0.0075):
    '''Calculate subset of points that make a convex hull around po

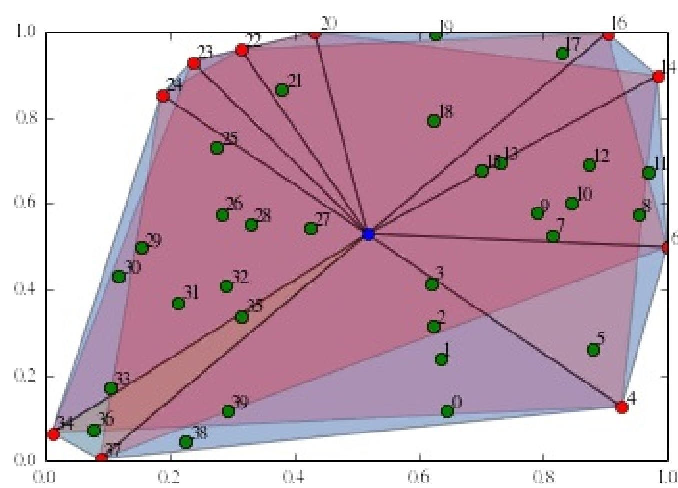
Recursively eliminates points that lie inside two neighbouring poi

:Parameters:
 points : ndarray (2 x m)
 array of points for which to find hull
 graphic : bool
 use pylab to show progress?
 smidgen : float
 offset for graphic number labels - useful values depend on your da

:Returns:
 hull_points : ndarray (2 x n)
 convex hull surrounding points
'''
    if graphic:
        p.clf()
        p.plot(points[0], points[1], 'ro')
    n_pts = points.shape[1]
    assert(n_pts > 5)
```

```
    centre = points.mean(1)
    if graphic: p.plot((centre[0],),(centre[1],),'bo')
    angles = n.apply_along_axis(_angle_to_point, 0, points, centre)
    pts_ord = points[:,angles.argsort()]
    if graphic:
        for i in xrange(n_pts):
            p.text(pts_ord[0,i] + smidgen, pts_ord[1,i] + smidgen, \
                   '%d' % i)
    pts = [x[0] for x in zip(pts_ord.transpose())]
    prev_pts = len(pts) + 1
    k = 0
    while prev_pts > n_pts:
        prev_pts = n_pts
        n_pts = len(pts)
        if graphic: p.gca().patches = []
        i = -2
        while i < (n_pts - 2):
            Aij = area_of_triangle(centre, pts[i],     pts[(i + 1)
            Ajk = area_of_triangle(centre, pts[(i + 1) % n_pts], \
                                   pts[(i + 2) % n_pts])
            Aik = area_of_triangle(centre, pts[i],     pts[(i + 2)
            if graphic:
                _draw_triangle(centre, pts[i], pts[(i + 1) % n_pts]
                               facecolor='blue', alpha = 0.2)
                _draw_triangle(centre, pts[(i + 1) % n_pts], \
                               pts[(i + 2) % n_pts], \
                               facecolor='green', alpha = 0.2)
                _draw_triangle(centre, pts[i], pts[(i + 2) % n_pts]
                               facecolor='red', alpha = 0.2)
            if Aij + Ajk < Aik:
                if graphic: p.plot((pts[i + 1][0],),(pts[i + 1][1],
                del pts[i+1]
            i += 1
            n_pts = len(pts)
        k += 1
    return n.asarray(pts)

if __name__ == "__main__":
    points = n.random.random_sample((2,40))
    hull_pts = convex_hull(points)
```



1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Finding the minimum point in the convex hull of a finite set of points

Based on the work of Philip Wolf [1] and the recursive algorithm of Kazuyuki Sekitani and Yoshitsugu Yamamoto [2].

The algorithm in [2] has 3 epsilon to avoid comparison problems in three parts of the algorithm. The code below has few changes and only one epsilon. The aim of the change is to avoid infinite loops.

Code

```
from numpy import array, matrix, sin, sqrt, dot, cos, ix_, zeros, c
from numpy.linalg import norm

from mpmath import mpf, mp
mp.dps=80

def find_min_point(P):
#    print "Calling find_min with P: ", P

    if len(P) == 1:
        return P[0]

    eps = mpf(10)**-40

    P = [array([mpf(i) for i in p]) for p in P]

    # Step 0\. Choose a point from C(P)
    x  = P[array([dot(p,p) for p in P]).argmin()]

    while True:

        # Step 1\. \alpha_k := min{x_{k-1}^T p | p \in P}
        p_alpha = P[array([dot(x,p) for p in P]).argmin()]

        if dot(x,x-p_alpha) < eps:
            return array([float(i) for i in x])

        Pk = [p for p in P if abs(dot(x,p-p_alpha)) < eps]

        # Step 2\. P_k := { p | p \in P and x_{k-1}^T p = \alpha_k]
        P_Pk = [p for p in P if not array([(p == q).all() for q in

        if len(Pk) == len(P):
            return array([float(i) for i in x])
```

```
        y = find_min_point(Pk)

        p_beta = P_Pk[array([dot(y,p) for p in P_Pk]).argmin()]

        if dot(y,y-p_beta) < eps:
            return array([float(i) for i in y])

        # Step 4.
        P_aux = [p for p in P_Pk if (dot(y-x,y-p)>eps) and (dot(x,y
        p_lambda = P_aux[array([dot(y,y-p)/dot(x,y-p) for p in P_au
        lam = dot(x,p_lambda-y) / dot(y-x,y-p_lambda)

        x += lam * (y-x)

if __name__ == '__main__':
    print find_min_point( [array([ -4.83907292e+00,   2.22438863e+0

    print find_min_point( [array([ -4.83907292e+00,   2.22438863e+0
```

```
[ 13.0643029   -3.03446491  -2.65980139]
[  1.61870596e-04  -3.78774039e-05  -3.29329552e-05]
```

References

1. Finding the nearest point in A polytope
2. A recursive algorithm for finding the minimum norm point in a polytope and a pair of closest points in two polytopes

Frequency swept signals

This page demonstrates two functions in scipy.signal for generating frequency-swept signals: `chirp` and `sweep_poly` .

Some of these require SciPy 0.8.

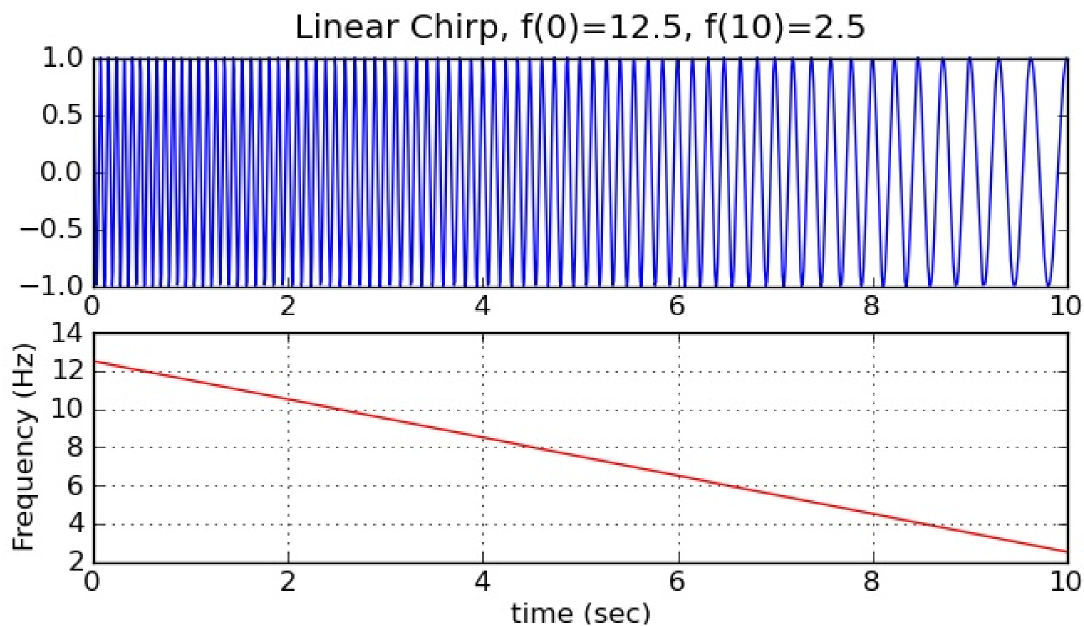
To run the code samples, you will need the following imports:

```
import numpy as np
from scipy.signal import chirp, sweep_poly
```

Linear Chirp

Sample code:

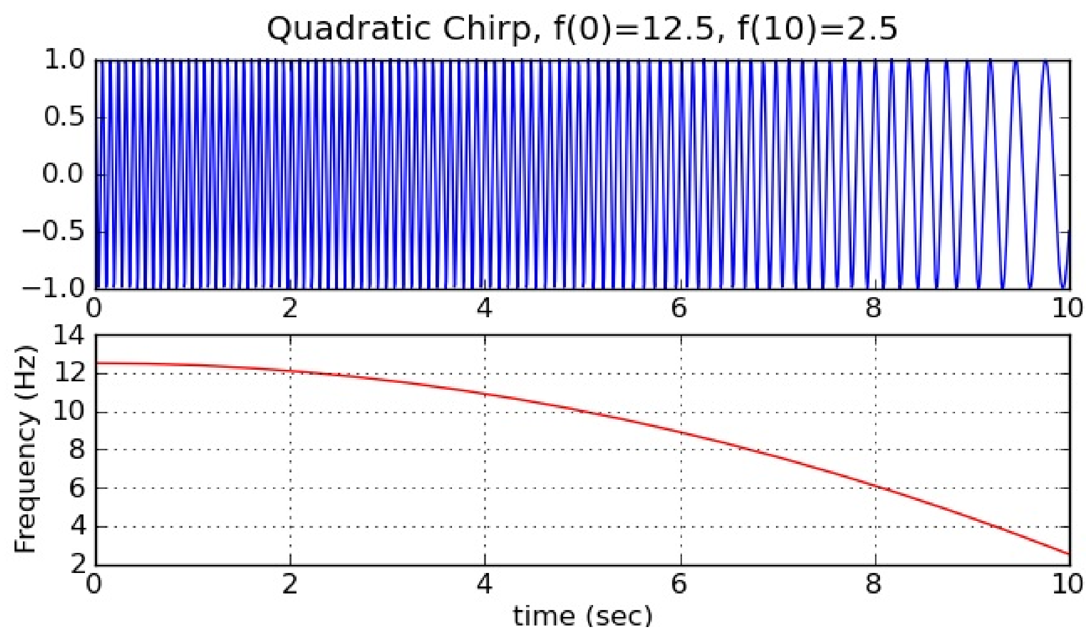
```
t = np.linspace(0, 10, 5001)
w = chirp(t, f0=12.5, f1=2.5, t1=10, method='linear')
```



Quadratic Chirp

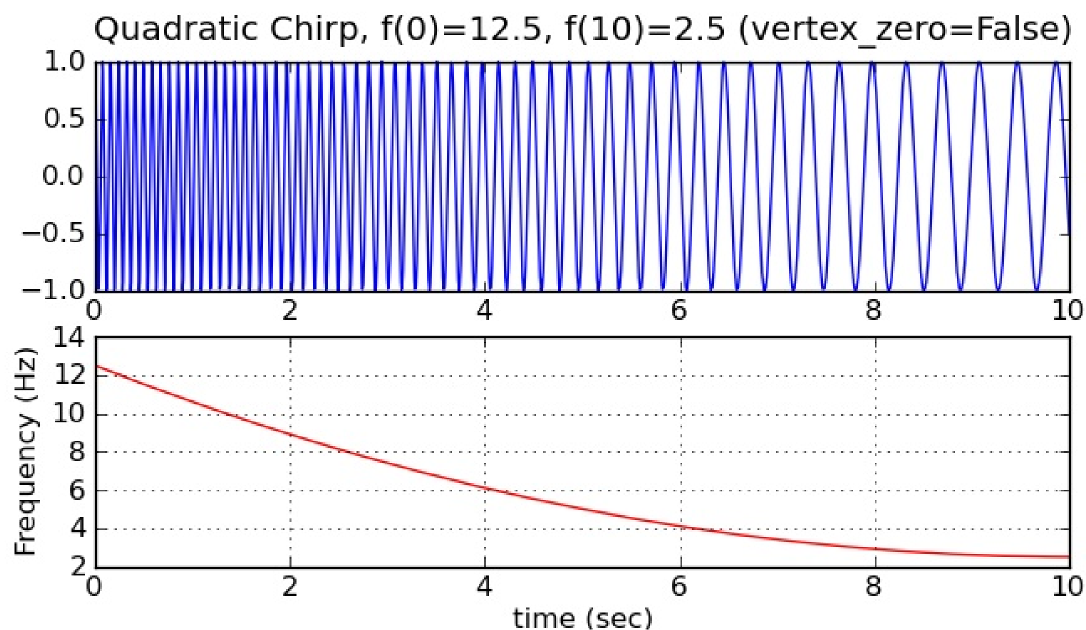
Sample code:

```
t = np.linspace(0, 10, 5001)
w = chirp(t, f0=12.5, f1=2.5, t1=10, method='quadratic')
```



Sample code using `vertex_zero` :

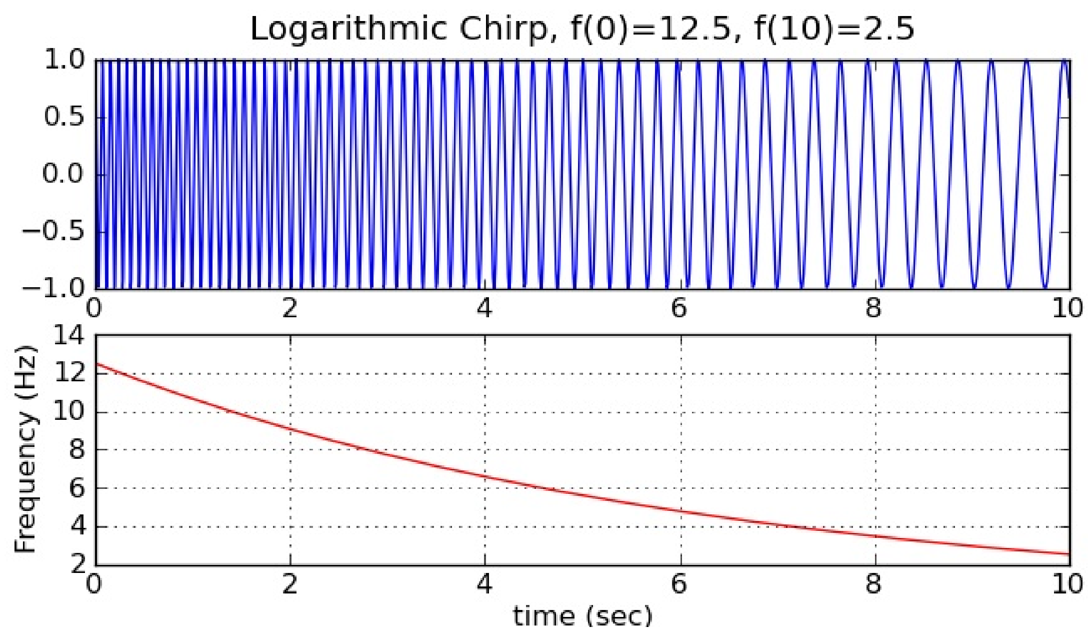
```
t = np.linspace(0, 10, 5001)
w = chirp(t, f0=12.5, f1=2.5, t1=10, method='quadratic', vertex_zer
```



Logarithmic Chirp

Sample code:

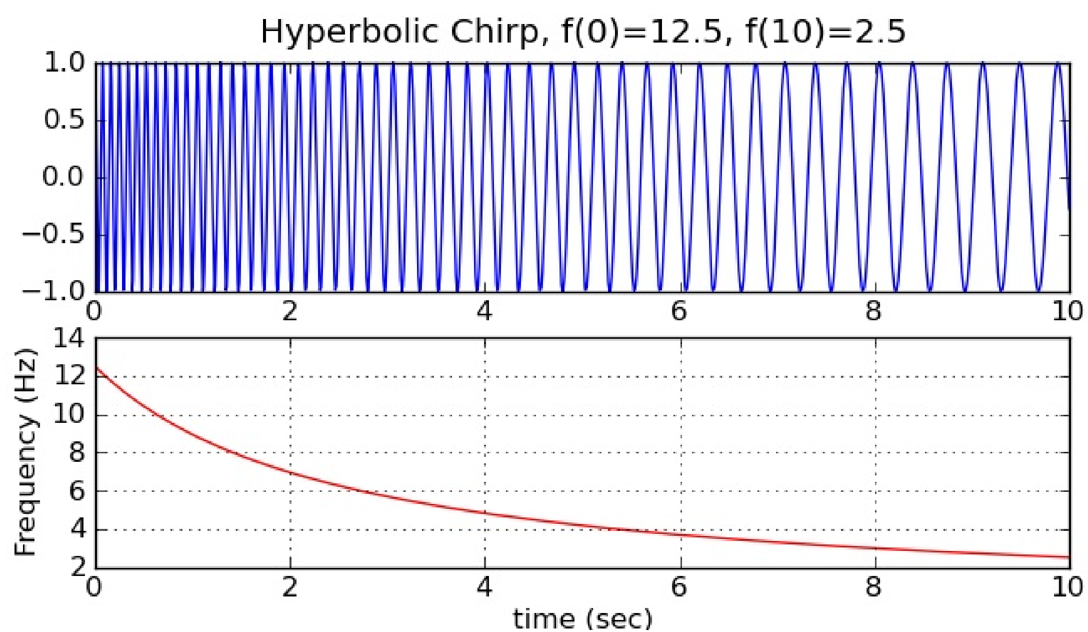
```
t = np.linspace(0, 10, 5001)
w = chirp(t, f0=12.5, f1=2.5, t1=10, method='logarithmic')
```



Hyperbolic Chirp

Sample code:

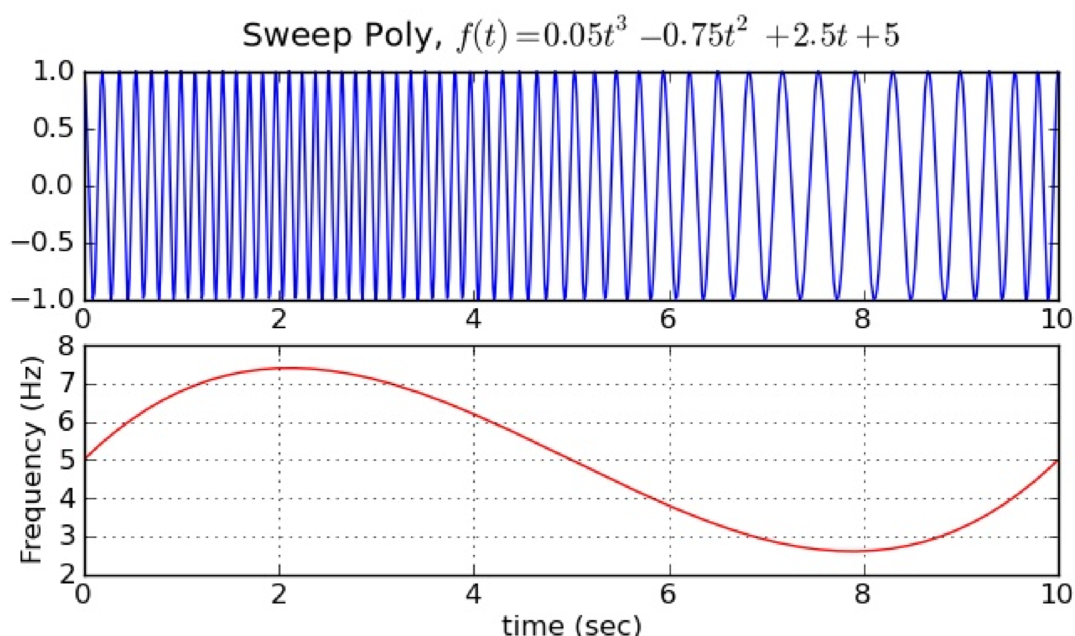
```
t = np.linspace(0, 10, 5001)
w = chirp(t, f0=12.5, f1=2.5, t1=10, method='hyperbolic')
```



Sweep Poly

Sample code:

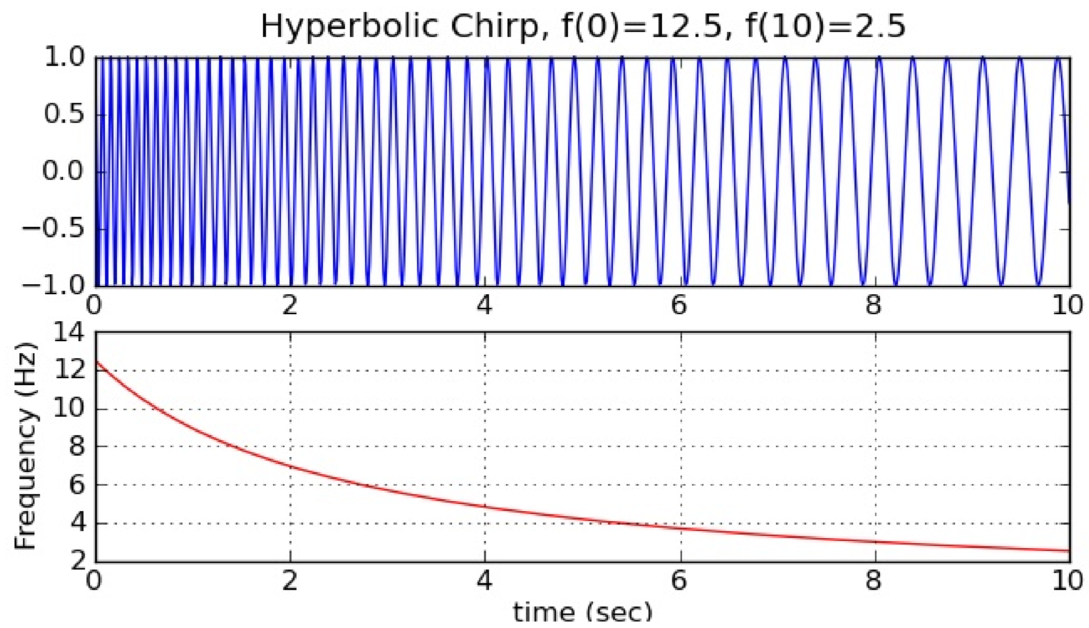
```
p = np.poly1d([0.05, -0.75, 2.5, 5.0])
t = np.linspace(0, 10, 5001)
w = sweep_poly(t, p)
```

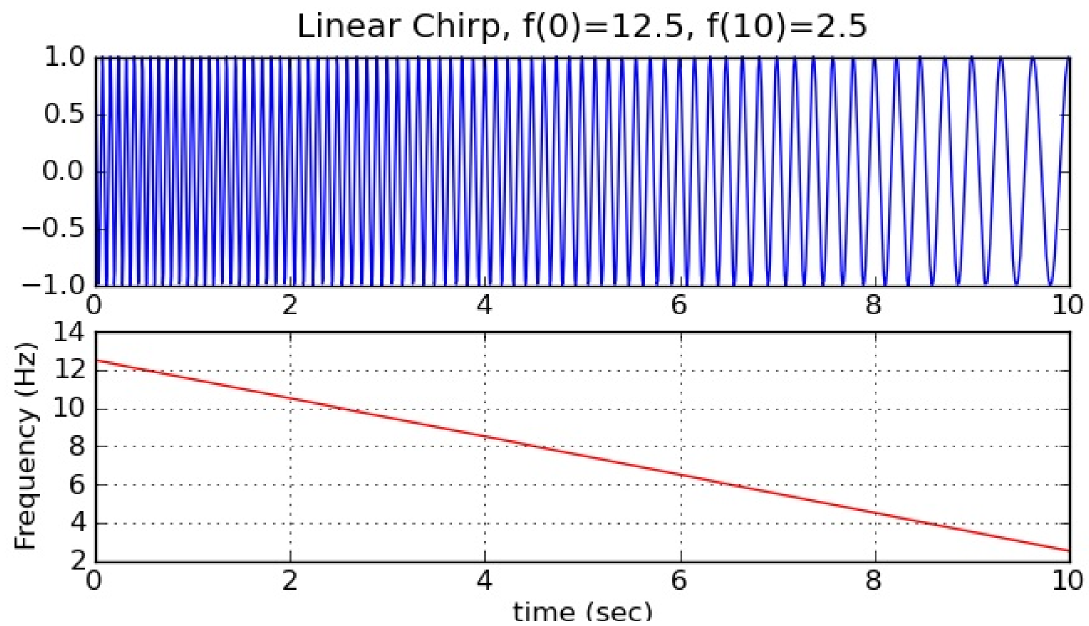


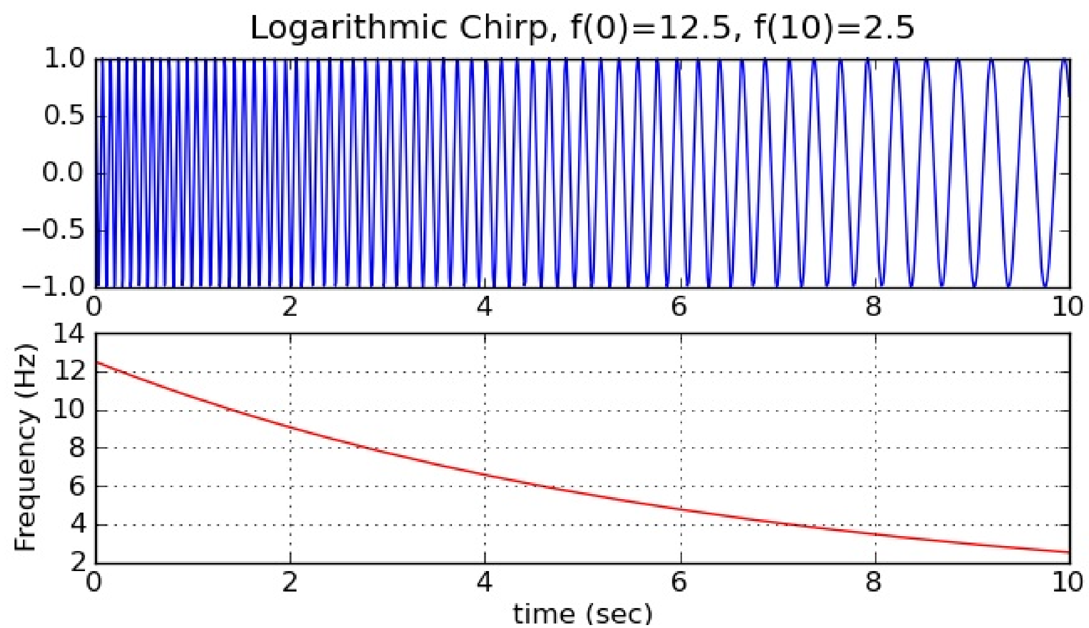
The script that generated the plots is `here`

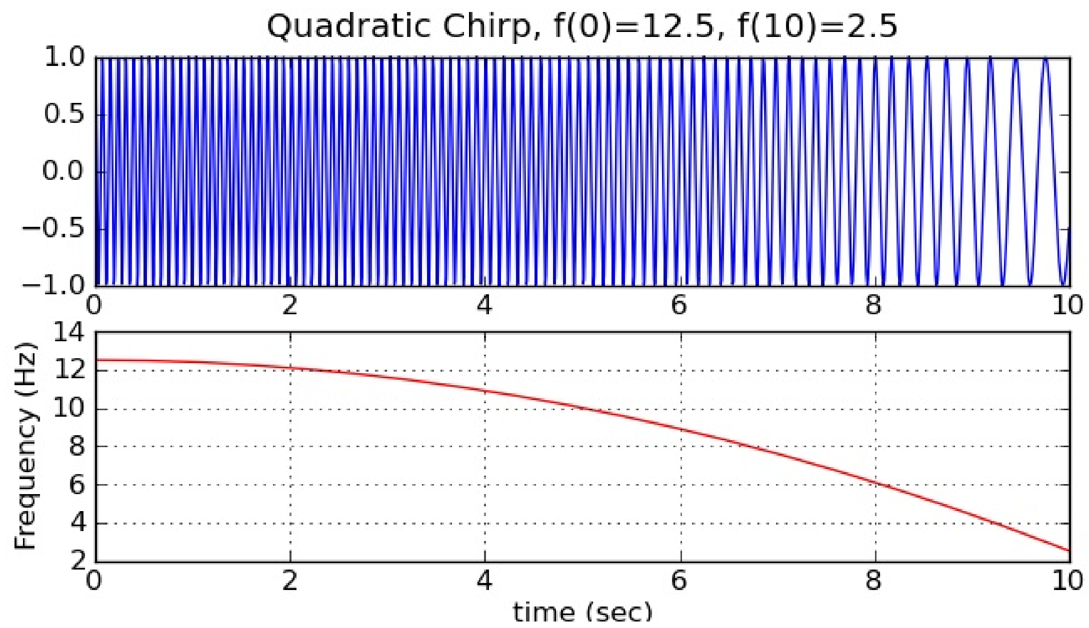
Attachments

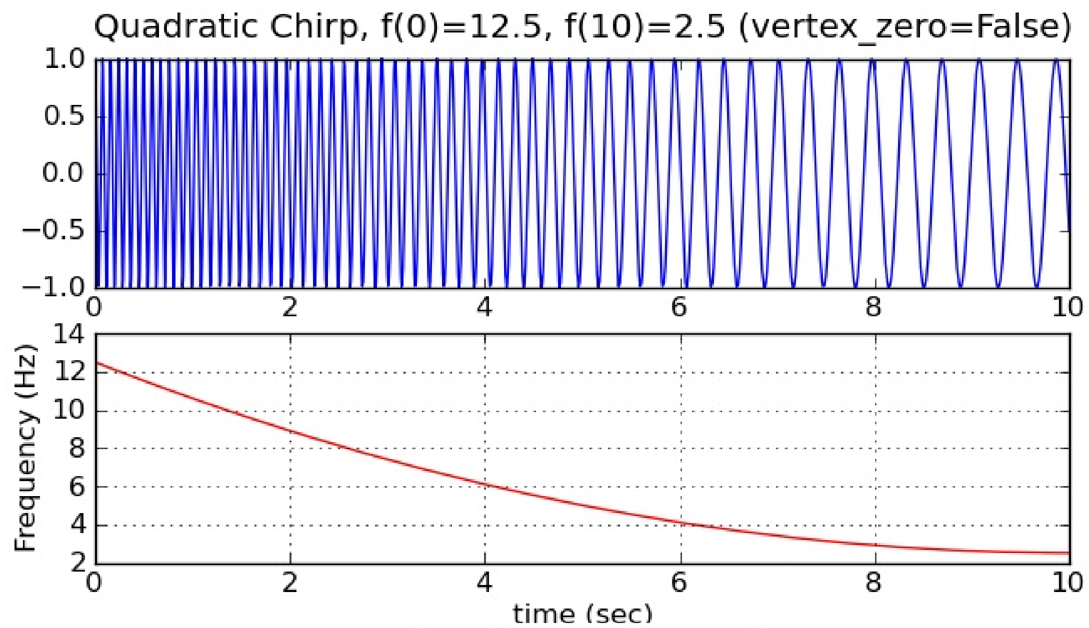
- `chirp_hyperbolic.png`
- `chirp_linear.png`
- `chirp_logarithmic.png`
- `chirp_plot.py`
- `chirp_quadratic.png`
- `chirp_quadratic_v0false.png`
- `sweep_poly.png`

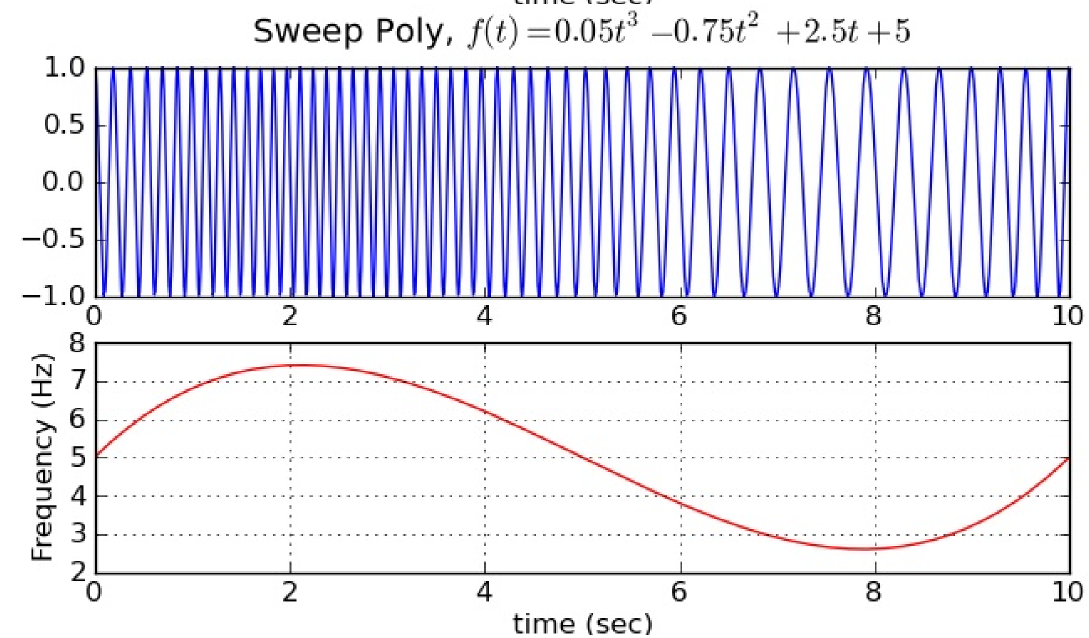

Hyperbolic Chirp, f(0)=12.5, f(10)=2.5
Frequency (Hz)
time (sec)


Linear Chirp, f(0)=12.5, f(10)=2.5
Frequency (Hz)
time (sec)


Logarithmic Chirp, f(0)=12.5, f(10)=2.5
Frequency (Hz)
time (sec)


Quadratic Chirp, f(0)=12.5, f(10)=2.5
Frequency (Hz)
time (sec)


Quadratic Chirp, f(0)=12.5, f(10)=2.5 (vertex_zero=False)
Frequency (Hz)
time (sec)


Sweep Poly, $f(t) = 0.05t^3 - 0.75t^2 + 2.5t + 5$
Frequency (Hz)
time (sec)

KDTree example

‘Note: there is an implementation of a kdtree in scipy: http://docs.scipy.org/scipy/docs/scipy.spatial.kdtree.KDTree/ It is recommended to use that instead of the below.‘

This is an example of how to construct and search a kd-tree in Pythonwith NumPy. kd-trees are e.g. used to search for neighbouring data points in multidimensional space. Searching the kd-tree for the nearest neighbour of all n points has O(n log n) complexity with respect to sample size.

Building a kd-tree

```
#!python numbers=disable

# Copyleft 2008 Sturla Molden
# University of Oslo

#import psyco
#psyco.full()

import numpy

def kdtree( data, leafsize=10 ):
    """
 build a kd-tree for O(n log n) nearest neighbour search

 input:
 data:        2D ndarray, shape =(ndim,ndata), preferentially C orde
 leafsize:    max. number of data points to leave in a leaf

 output:
 kd-tree:     list of tuples
 """

    ndim = data.shape[0]
    ndata = data.shape[1]

    # find bounding hyper-rectangle
    hrect = numpy.zeros((2,data.shape[0]))
    hrect[0,:] = data.min(axis=1)
    hrect[1,:] = data.max(axis=1)

    # create root of kd-tree
    idx = numpy.argsort(data[0,:], kind='mergesort')
    data[:,:] = data[:,idx]
    splitval = data[0,ndata/2]
```

```
    left_hrect = hrect.copy()
    right_hrect = hrect.copy()
    left_hrect[1, 0] = splitval
    right_hrect[0, 0] = splitval

    tree = [(None, None, left_hrect, right_hrect, None, None)]

    stack = [(data[:,:ndata/2], idx[:ndata/2], 1, 0, True),
             (data[:,ndata/2:], idx[ndata/2:], 1, 0, False)]

    # recursively split data in halves using hyper-rectangles:
    while stack:

        # pop data off stack
        data, didx, depth, parent, leftbranch = stack.pop()
        ndata = data.shape[1]
        nodeptr = len(tree)

        # update parent node

        _didx, _data, _left_hrect, _right_hrect, left, right = tree

        tree[parent] = (_didx, _data, _left_hrect, _right_hrect, no
            else (_didx, _data, _left_hrect, _right_hrect, left, no

        # insert node in kd-tree

        # leaf node?
        if ndata <= leafsize:
            _didx = didx.copy()
            _data = data.copy()
            leaf = (_didx, _data, None, None, 0, 0)
            tree.append(leaf)

        # not a leaf, split the data in two
        else:
            splitdim = depth % ndim
            idx = numpy.argsort(data[splitdim,:], kind='mergesort')
            data[:,:] = data[:,idx]
            didx = didx[idx]
            nodeptr = len(tree)
            stack.append((data[:,:ndata/2], didx[:ndata/2], depth+1
            stack.append((data[:,ndata/2:], didx[ndata/2:], depth+1
            splitval = data[splitdim,ndata/2]
            if leftbranch:
                left_hrect = _left_hrect.copy()
                right_hrect = _left_hrect.copy()
            else:
                left_hrect = _right_hrect.copy()
                right_hrect = _right_hrect.copy()
            left_hrect[1, splitdim] = splitval
            right_hrect[0, splitdim] = splitval
```

```
            # append node to tree
            tree.append((None, None, left_hrect, right_hrect, None,

    return tree
```

Searching a kd-tree

```
#!python numbers=disable

def intersect(hrect, r2, centroid):
    """
 checks if the hyperrectangle hrect intersects with the
 hypersphere defined by centroid and r2
 """
    maxval = hrect[1,:]
    minval = hrect[0,:]
    p = centroid.copy()
    idx = p < minval
    p[idx] = minval[idx]
    idx = p > maxval
    p[idx] = maxval[idx]
    return ((p-centroid)**2).sum() < r2

def quadratic_knn_search(data, lidx, ldata, K):
    """ find K nearest neighbours of data among ldata """
    ndata = ldata.shape[1]
    param = ldata.shape[0]
    K = K if K < ndata else ndata
    retval = []
    sqd = ((ldata - data[:,:ndata])**2).sum(axis=0) # data.reshape(
    idx = numpy.argsort(sqd, kind='mergesort')
    idx = idx[:K]
    return zip(sqd[idx], lidx[idx])

def search_kdtree(tree, datapoint, K):
    """ find the k nearest neighbours of datapoint in a kdtree """
    stack = [tree[0]]
    knn = [(numpy.inf, None)]*K
    _datapt = datapoint[:,0]
    while stack:

        leaf_idx, leaf_data, left_hrect, \
                  right_hrect, left, right = stack.pop()

        # leaf
        if leaf_idx is not None:
            _knn = quadratic_knn_search(datapoint, leaf_idx, leaf_d
            if _knn[0][0] < knn[-1][0]:
                knn = sorted(knn + _knn)[:K]
```

```
        # not a leaf
        else:

            # check left branch
            if intersect(left_hrect, knn[-1][0], _datapt):
                stack.append(tree[left])

            # chech right branch
            if intersect(right_hrect, knn[-1][0], _datapt):
                stack.append(tree[right])
    return knn

def knn_search( data, K, leafsize=2048 ):

    """ find the K nearest neighbours for data points in data,
 using an O(n log n) kd-tree """

    ndata = data.shape[1]
    param = data.shape[0]

    # build kdtree
    tree = kdtree(data.copy(), leafsize=leafsize)

    # search kdtree
    knn = []
    for i in numpy.arange(ndata):
        _data = data[:,i].reshape((param,1)).repeat(leafsize, axis=
        _knn = search_kdtree(tree, _data, K+1)
        knn.append(_knn[1:])

    return knn

def radius_search(tree, datapoint, radius):
    """ find all points within radius of datapoint """
    stack = [tree[0]]
    inside = []
    while stack:

        leaf_idx, leaf_data, left_hrect, \
                  right_hrect, left, right = stack.pop()

        # leaf
        if leaf_idx is not None:
            param=leaf_data.shape[0]
            distance = numpy.sqrt(((leaf_data - datapoint.reshape((
            near = numpy.where(distance<=radius)
            if len(near[0]):
                idx = leaf_idx[near]
                distance = distance[near]
                inside += (zip(distance, idx))

        else:
```

```
            if intersect(left_hrect, radius, datapoint):
                stack.append(tree[left])

            if intersect(right_hrect, radius, datapoint):
                stack.append(tree[right])

    return inside
```

Quadratic search for small data sets

In contrast to the kd-tree, straight forward exhaustive search has quadratic complexity with respect to sample size. It can be faster than using a kd-tree when the sample size is very small. On my computer that is approximately 500 samples or less.

```
#!python numbers=disable

def knn_search( data, K ):
    """ find the K nearest neighbours for data points in data,
 using O(n**2) search """
    ndata = data.shape[1]
    knn = []
    idx = numpy.arange(ndata)
    for i in numpy.arange(ndata):
        _knn = quadratic_knn_search(data[:,i], idx, data, K+1) # se
        knn.append( _knn[1:] )
    return knn
```

Parallel search for large data sets

While creating a kd-tree is very fast, searching it can be time consuming. Due to Python's dreaded "Global Interpreter Lock" (GIL), threads cannot be used to conduct multiple searches in parallel. That is, Python threads can be used for asynchrony but not concurrency. However, we can use multiple processes (multiple interpreters). The pyprocessing package makes this easy. It has an API similar to Python's threading and Queue standard modules, but work with processes instead of threads. Beginning with Python 2.6, pyprocessing is already included in Python's standard library as the "multiprocessing" module. There is a small overhead of using multiple processes, including process creation, process startup, IPC, and process termination. However, because processes run in separate address spaces, no memory contention is incurred. In the following example, the overhead of using multiple processes is very small compared to the computation, giving a speed-up close to the number of CPUs on the computer.

```
#!python numbers=disable

try:
    import multiprocessing as processing
except:
    import processing

import ctypes, os

def __num_processors():
    if os.name == 'nt': # Windows
        return int(os.getenv('NUMBER_OF_PROCESSORS'))
    else: # glibc (Linux, *BSD, Apple)
        get_nprocs = ctypes.cdll.libc.get_nprocs
        get_nprocs.restype = ctypes.c_int
        get_nprocs.argtypes = []
        return get_nprocs()

def __search_kdtree(tree, data, K, leafsize):
    knn = []
    param = data.shape[0]
    ndata = data.shape[1]
    for i in numpy.arange(ndata):
        _data = data[:,i].reshape((param,1)).repeat(leafsize, axis=
        _knn = search_kdtree(tree, _data, K+1)
        knn.append(_knn[1:])
    return knn

def __remote_process(rank, qin, qout, tree, K, leafsize):
    while 1:
        # read input queue (block until data arrives)
        nc, data = qin.get()
        # process data
        knn = __search_kdtree(tree, data, K, leafsize)
        # write to output queue
        qout.put((nc,knn))

def knn_search_parallel(data, K, leafsize=2048):

    """ find the K nearest neighbours for data points in data,
 using an O(n log n) kd-tree, exploiting all logical
 processors on the computer """

    ndata = data.shape[1]
    param = data.shape[0]
    nproc = __num_processors()
    # build kdtree
    tree = kdtree(data.copy(), leafsize=leafsize)
    # compute chunk size
    chunk_size = data.shape[1] / (4*nproc)
    chunk_size = 100 if chunk_size < 100 else chunk_size
    # set up a pool of processes
```

```
    qin = processing.Queue(maxsize=ndata/chunk_size)
    qout = processing.Queue(maxsize=ndata/chunk_size)
    pool = [processing.Process(target=__remote_process,
                args=(rank, qin, qout, tree, K, leafsize))
                    for rank in range(nproc)]
    for p in pool: p.start()
    # put data chunks in input queue
    cur, nc = 0, 0
    while 1:
        _data = data[:,cur:cur+chunk_size]
        if _data.shape[1] == 0: break
        qin.put((nc,_data))
        cur += chunk_size
        nc += 1
    # read output queue
    knn = []
    while len(knn) < nc:
        knn += [qout.get()]
    # avoid race condition
    _knn = [n for i,n in sorted(knn)]
    knn = []
    for tmp in _knn:
        knn += tmp
    # terminate workers
    for p in pool: p.terminate()
    return knn
```

Running the code

The following shows how to run the example code (including how input data should be formatted):

```
#!python numbers=disable

from time import clock

def test():
    K = 11
    ndata = 10000
    ndim = 12
    data =  10 * numpy.random.rand(ndata*ndim).reshape((ndim,ndata)
    knn_search(data, K)

if __name__ == '__main__':
    t0 = clock()
    test()
    t1 = clock()
    print "Elapsed time %.2f seconds" % t1-t0

    #import profile          # using Python's profiler is not usefu
    #profile.run('test()')   # running the parallel search.
```

Kalman filtering

This is code implements the example given in pages 11-15 of An Introduction to the Kalman Filter by Greg Welch and Gary Bishop, University of North Carolina at Chapel Hill, Department of Computer Science.

```
# Kalman filter example demo in Python

# A Python implementation of the example given in pages 11-15 of "A
# Introduction to the Kalman Filter" by Greg Welch and Gary Bishop,
# University of North Carolina at Chapel Hill, Department of Comput
# Science, TR 95-041,
# http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html

# by Andrew D. Straw

import numpy as np
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = (10, 8)

# intial parameters
n_iter = 50
sz = (n_iter,) # size of array
x = -0.37727 # truth value (typo in example at top of p. 13 calls t
z = np.random.normal(x,0.1,size=sz) # observations (normal about x,

Q = 1e-5 # process variance

# allocate space for arrays
xhat=np.zeros(sz)      # a posteri estimate of x
P=np.zeros(sz)         # a posteri error estimate
xhatminus=np.zeros(sz) # a priori estimate of x
Pminus=np.zeros(sz)    # a priori error estimate
K=np.zeros(sz)         # gain or blending factor

R = 0.1**2 # estimate of measurement variance, change to see effect

# intial guesses
xhat[0] = 0.0
P[0] = 1.0

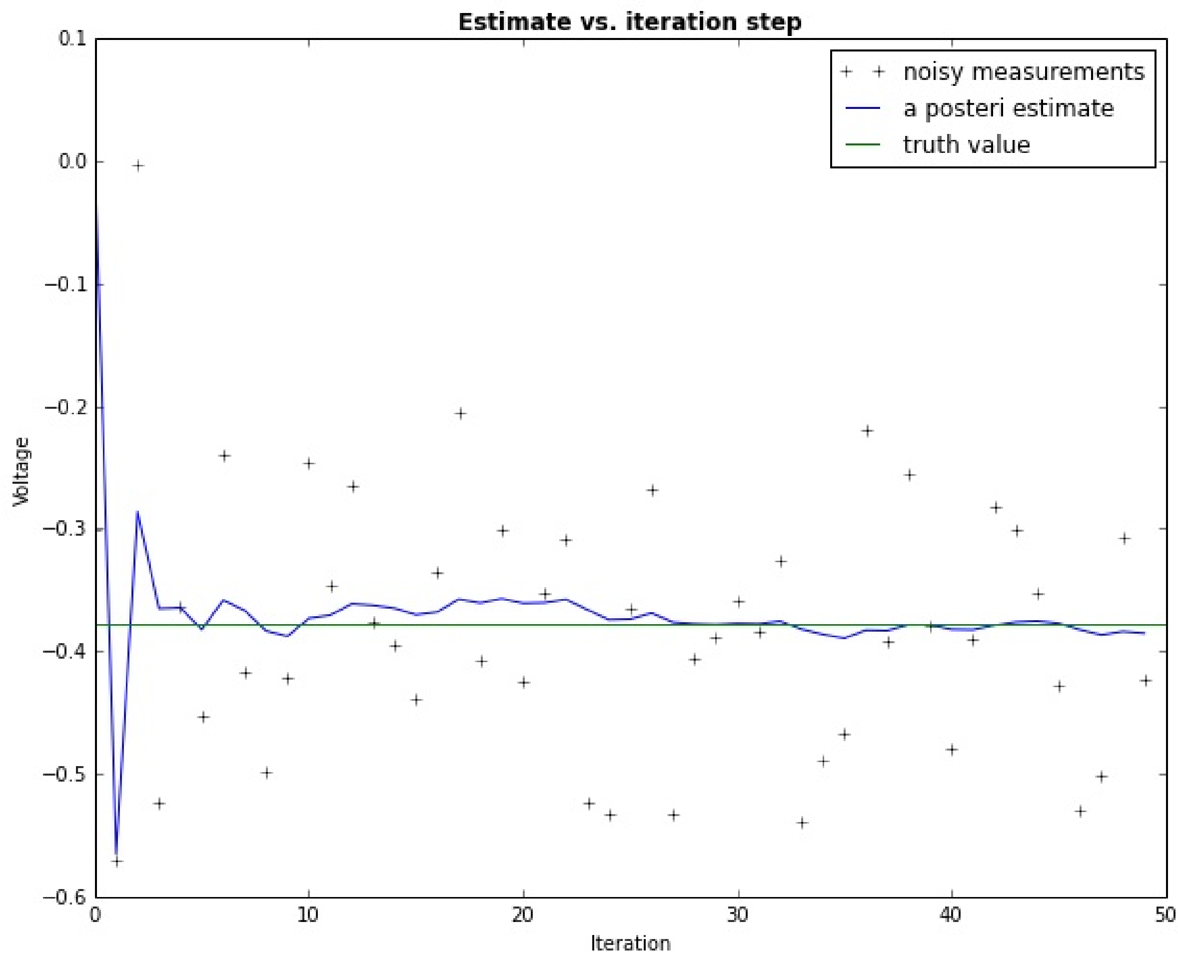
for k in range(1,n_iter):
    # time update
    xhatminus[k] = xhat[k-1]
    Pminus[k] = P[k-1]+Q

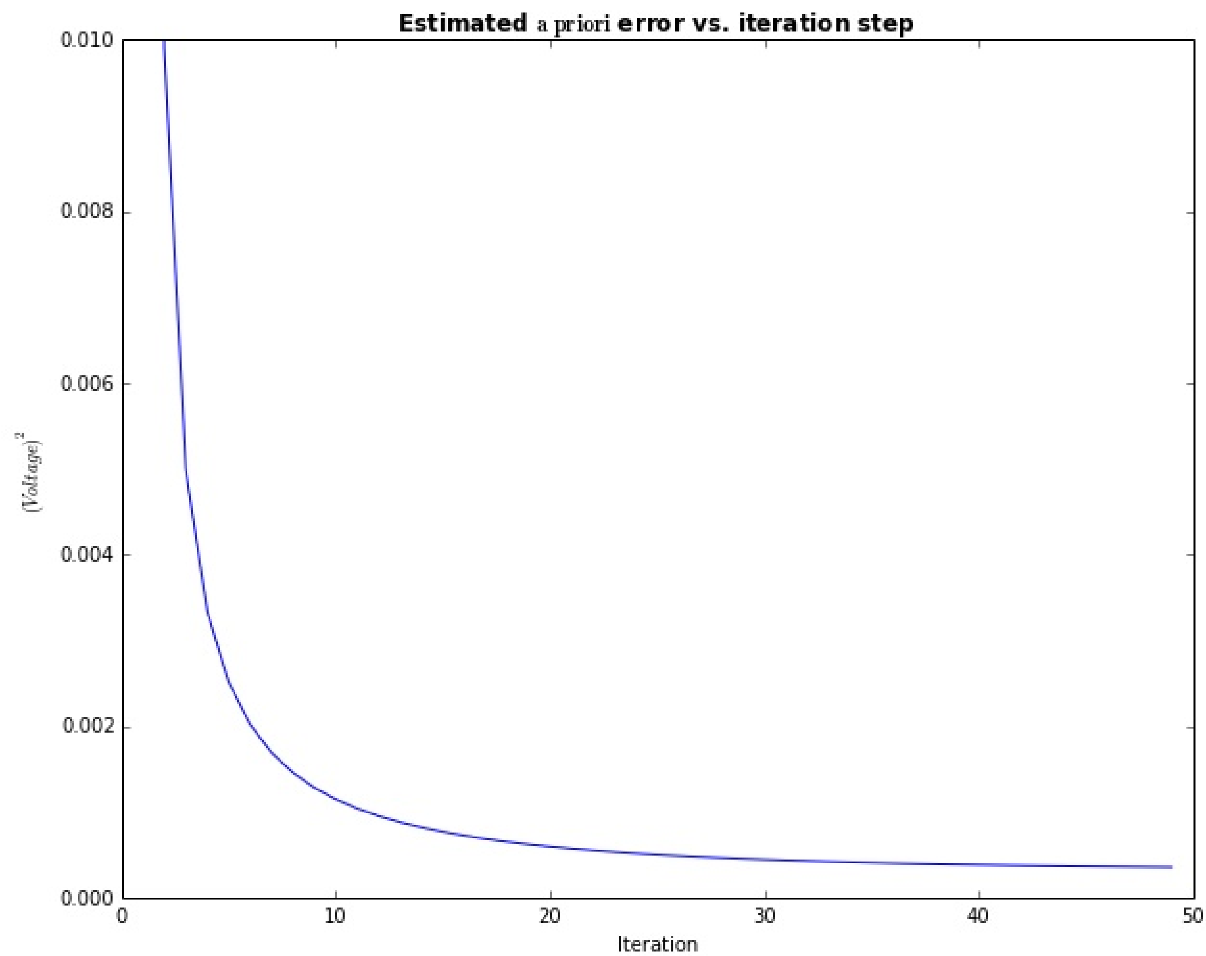
    # measurement update
    K[k] = Pminus[k]/( Pminus[k]+R )
```

```
    xhat[k] = xhatminus[k]+K[k]*(z[k]-xhatminus[k])
    P[k] = (1-K[k])*Pminus[k]

plt.figure()
plt.plot(z,'k+',label='noisy measurements')
plt.plot(xhat,'b-',label='a posteri estimate')
plt.axhline(x,color='g',label='truth value')
plt.legend()
plt.title('Estimate vs. iteration step', fontweight='bold')
plt.xlabel('Iteration')
plt.ylabel('Voltage')

plt.figure()
valid_iter = range(1,n_iter) # Pminus not valid at step 0
plt.plot(valid_iter,Pminus[valid_iter],label='a priori error estima
plt.title('Estimated $\it{\mathbf{a \ priori}}$ error vs. iteration
plt.xlabel('Iteration')
plt.ylabel('$(Voltage)^2$')
plt.setp(plt.gca(),'ylim',[0,.01])
plt.show()
```


Estimate vs. iteration step
noisy measurements
a posteri estimate
truth value
Voltage
Iteration
0.1
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
0
10
20
30
40
50


Estimated a priori error vs. iteration step
(Voltage)²
Iteration
0.010
0.008
0.006
0.004
0.002
0.000
0
10
20
30
40
50

Linear classification

Fisher's Linear Discriminant

The first example shows the implementation of Fisher's Linear Classifier for 2-class problem and this algorithm is precisely described in book "Pattern Recognition and Machine Learning" by Christopher M Bishop (p 186, Section 4.1). The main idea of this algorithm is that we try to reduce the dimensionality of input vector X and project it onto 1D space using the equation y=W.T X where W.T - row vector of weights, and we adjust the weight vector W and choose the projection that maximizes the class separation. The following program use the famouse data set Iris with 150 number of instances and 4 attributes (4D space), target vector which contains labels: "Iris-setosa", "Iris-virginica", "Iris-versicolor", therefore, we have 3 classes, but, in this case, we may assume that we have class 1 with labels "Iris-setosa" and class 2 with other instances. Iris data set is available here: http://archive.ics.uci.edu/ml/datasets/Iris/ or here (comma separated format) - `bezdekIris.data.txt`

```
#! python
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt

def read_data():
  f=open("Iris.txt", 'r')
  lines=[line.strip() for line in f.readlines()]
  f.close()

  lines=[line.split(",") for line in lines if line]

  class1=np.array([line[:4] for line in lines if line[-1]=="Iris-se

  class2=np.array([line[:4] for line in lines if line[-1]!="Iris-se

  return class1, class2

def main():

  class1, class2=read_data()

  mean1=np.mean(class1, axis=0)
  mean2=np.mean(class2, axis=0)

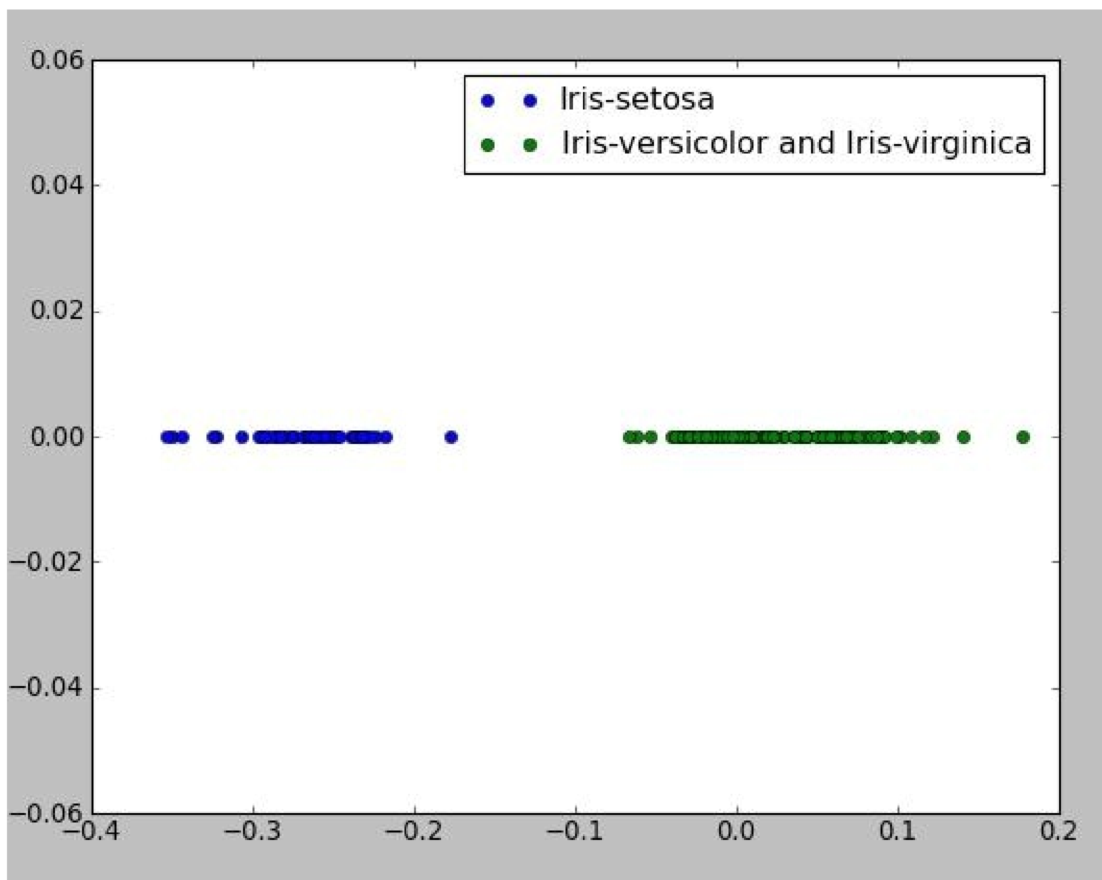
  #calculate variance within class
  Sw=np.dot((class1-mean1).T, (class1-mean1))+np.dot((class2-mean2)

  #calculate weights which maximize linear separation
  w=np.dot(np.linalg.inv(Sw), (mean2-mean1))

  print "vector of max weights", w
  #projection of classes on 1D space
  plt.plot(np.dot(class1, w), [0]*class1.shape[0], "bo", label="Iri
  plt.plot(np.dot(class2, w), [0]*class2.shape[0], "go", label="Iri
  plt.legend()

  plt.show()

main()
```



Probabilistic Generative Model

This program is the implementation of Probabilistic Generative Model for K-class problem which is also described in book “Pattern Recognition and Machine Learning” by Christopher M Bishop (p 196, Section 4.2). We try to learn the class-conditional densities (likelihood) p(x|Ck) for each class K, and prior probability density p(Ck), then we can compute posterior probability p(Ck|x) by using Bayes rule. Here we assume that p(x|Ck) are 4D Gaussians with parameters uk - mean vector of class K, Sk - covariance matrix of class K, also p(Ck) for all k is 1/3. Then we compute so called quantities ak (variables pc’s in the program) and if ak>>aj for all k!=j then assign p(Ck|x)=1 and p(Cj|x)=0.

```
#! python
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import math

def read_data():
  f=open("Iris.txt", 'r')
  lines=[line.strip() for line in f.readlines()]
  f.close()

  lines=[line.split(",") for line in lines if line]
```

```
    data=np.array([line[:4] for line in lines if line], dtype=np.floa

    class1=np.array([line[:4] for line in lines if line[-1]=="Iris-se

    class2=np.array([line[:4] for line in lines if line[-1]=="Iris-vi

    class3=np.array([line[:4] for line in lines if line[-1]=="Iris-ve

   #list of class labels
    labels=[]
    for line in lines:
     strt=line.pop()
     labels.append(strt)
    #create array of labels
    labels=[line.split(",") for line in labels if line]
    t=np.zeros(shape=(150, 3))
    #create target vector encoded according to 1-of-K scheme
    for i in xrange(len(data)):
     if labels[i]==["Iris-setosa"]: t[i][0]=1
     elif labels[i]==["Iris-versicolor"]: t[i][1]=1
     elif labels[i]==["Iris-virginica"]: t[i][2]=1

    return class1, class2, class3, data, t

  def gaussian(x, mean, cov):
    xm=np.reshape((x-mean), (-1, 1))
    px=1/(math.pow(2.0*math.pi, 2))*1/math.sqrt(np.linalg.det(cov))*n
    return px

  def main():
   class1, class2, class3, data, t=read_data()

   count=np.zeros(shape=(150,1))
   t_assigned=np.zeros(shape=(150, 3))
   cov=np.zeros(shape=(3, 4, 4))
   mean=np.zeros(shape=(3, 4))

    #compute means for each class
   mean1=class1.mean(axis=0)
   mean2=class2.mean(axis=0)
   mean3=class3.mean(axis=0)
    #compute covariance matrices, such that the columns are variables
   cov1=np.cov(class1, rowvar=0)
   cov2=np.cov(class2, rowvar=0)
   cov3=np.cov(class3, rowvar=0)

  #compute gaussian likelihood functions p(x|Ck) for each class
   for i in xrange(len(data)):
       px1=(1/3.0)*gaussian(data[i], mean1, cov1)
       px2=(1/3.0)*gaussian(data[i], mean2, cov2)
       px3=(1/3.0)*gaussian(data[i], mean3, cov3)
       m=np.max([px1, px2, px3])
   #compute posterior probability p(Ck|x) assuming that p(x|Ck) is ga
```

```
    pc1=((math.exp(px1)*math.exp(-m))*math.exp(m))/((math.exp(px2)
    pc2=((math.exp(px2)*math.exp(-m))*math.exp(m))/((math.exp(px1)
    pc3=((math.exp(px3)*math.exp(-m))*math.exp(m))/((math.exp(px1)
#assign p(Ck|x)=1 if p(Ck|x)>>p(Cj|x) for all j!=k
    if pc1>pc2 and pc1>pc3: t_assigned[i][0]=1
    elif pc3>pc1 and pc3>pc2: t_assigned[i][1]=1
    elif pc2>pc1 and pc2>pc3: t_assigned[i][2]=1
#count the number of misclassifications
    for j in xrange(3):
     if t[i][j]-t_assigned[i][j]!=0: count[i]=1

cov=[cov1, cov2, cov3]
mean=[mean1, mean2, mean3]

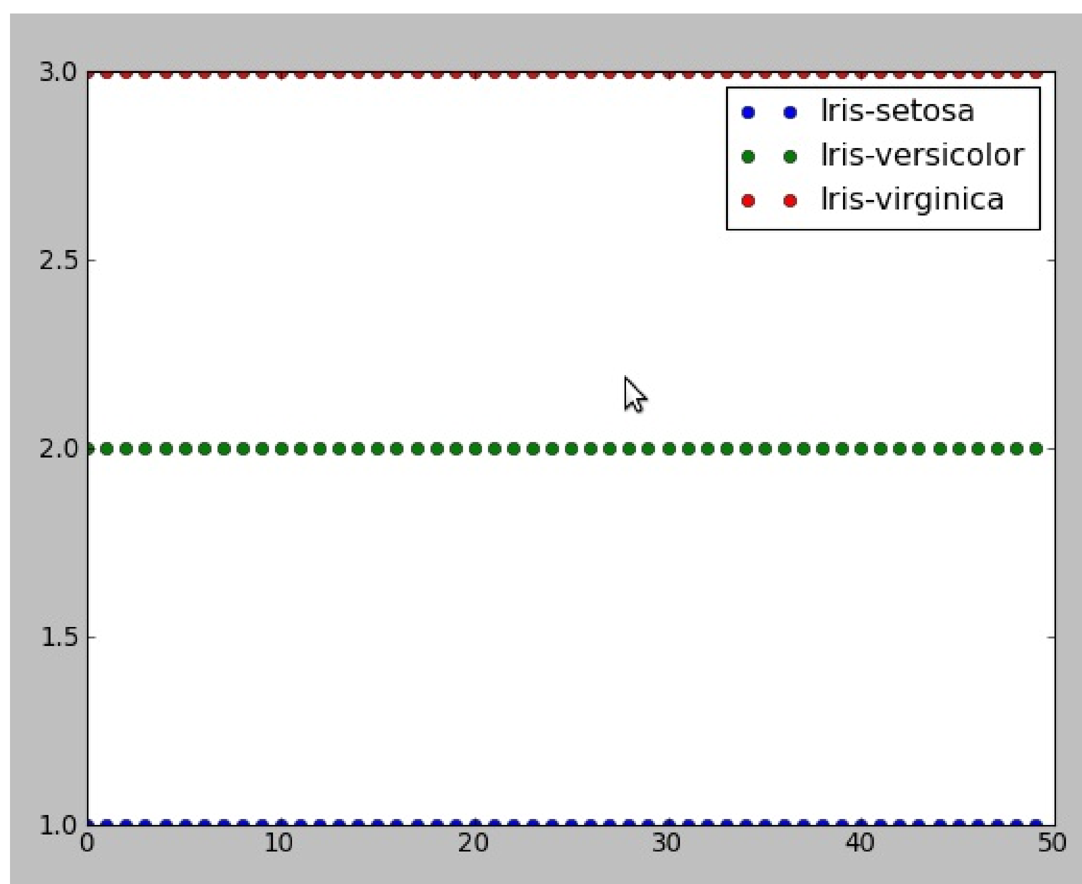
t1=np.zeros(shape=(len(class1), 1))
t2=np.zeros(shape=(len(class2), 1))
t3=np.zeros(shape=(len(class3), 1))
for i in xrange(len(data)):
 for j in xrange(len(class1)):
  if t_assigned[i][0]==1: t1[j]=1
  elif t_assigned[i][1]==1: t2[j]=2
  elif t_assigned[i][2]==1: t3[j]=3

plt.plot(t1, "bo", label="Iris-setosa")
plt.plot(t2, "go", label="Iris-versicolor")
plt.plot(t3, "ro", label="Iris-virginica")
plt.legend()
plt.show()

print "number of misclassifications", sum(count), "assigned labels

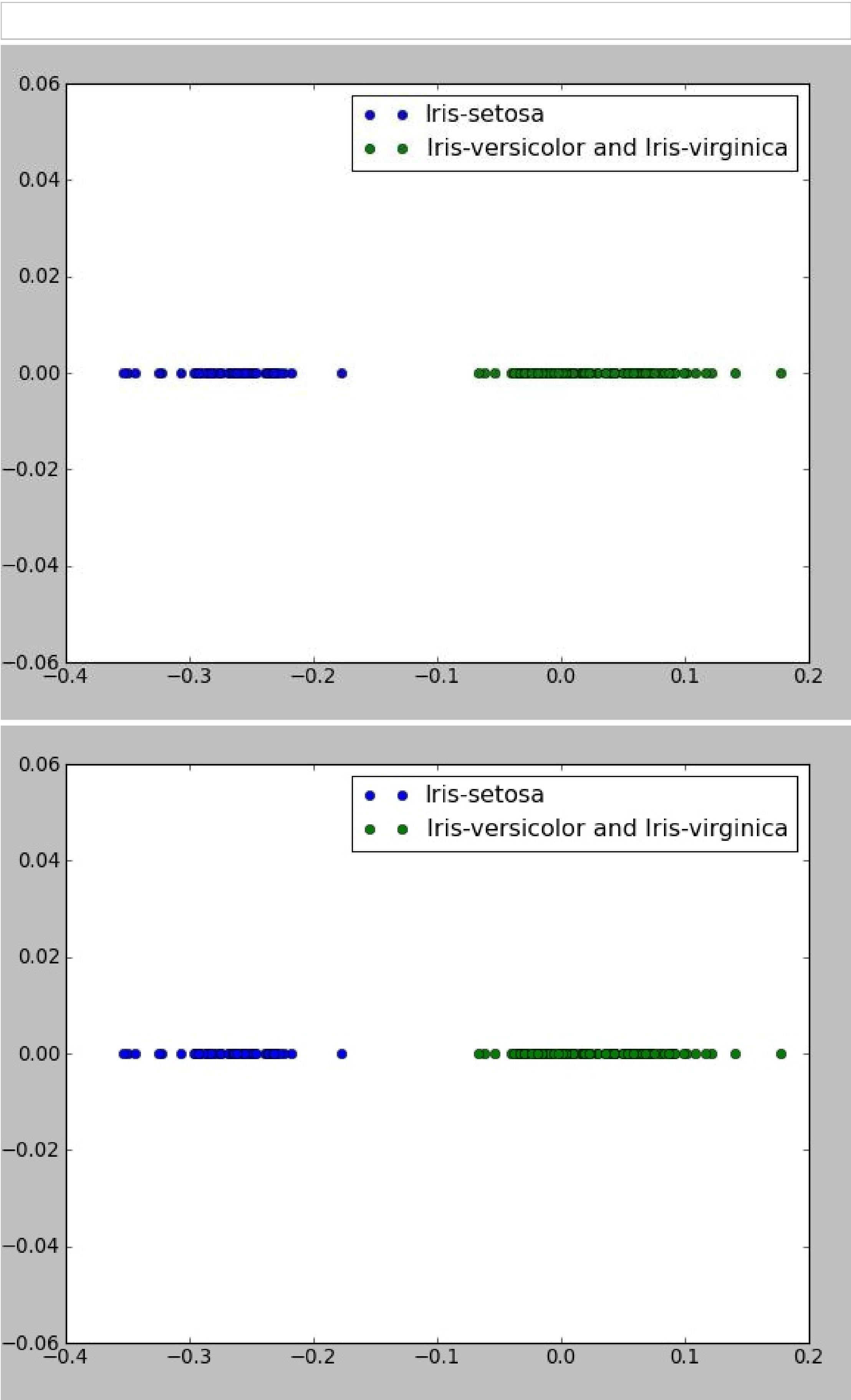
main()
```

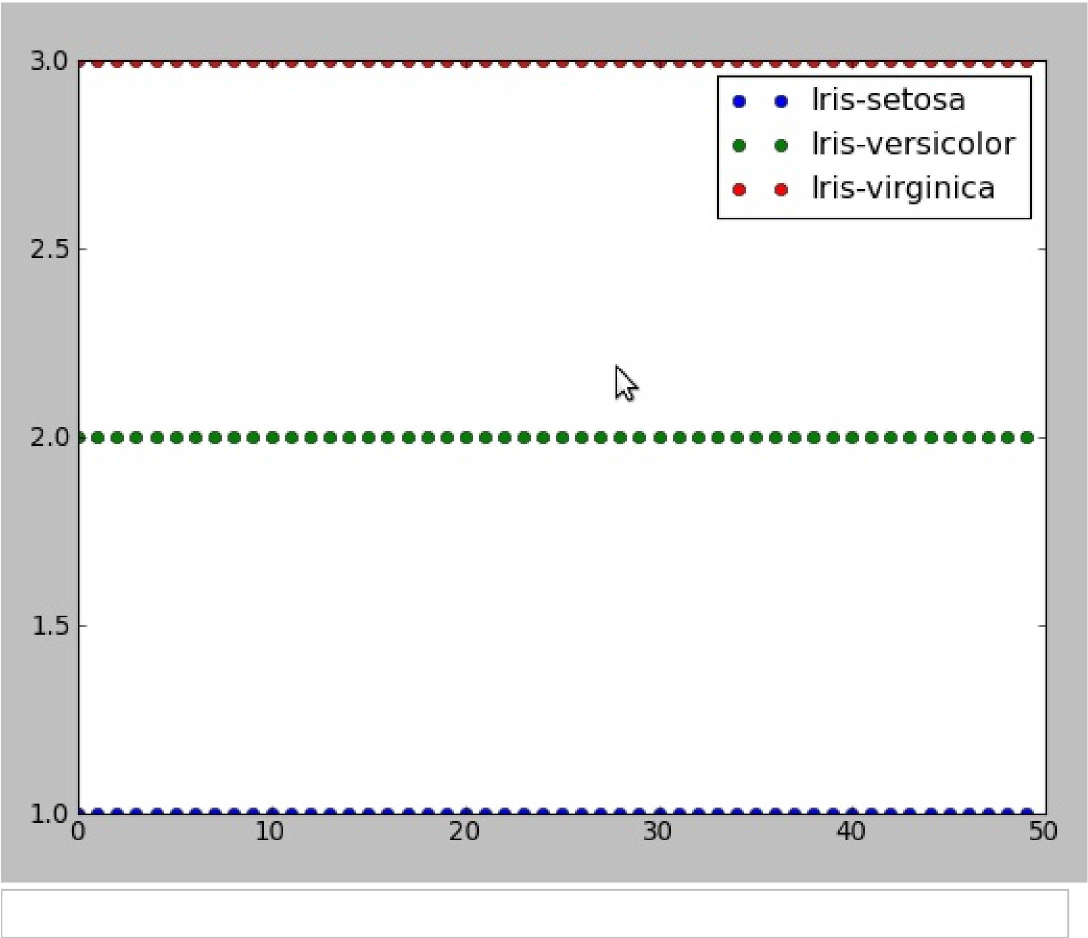
This program resulted in the number of misclassifications = 3 out of all 150 instances



Attachments

- `Fisher_discriminant.png`
- `Fisher_disrciminant.JPG`
- `Fisher_disrciminant.PNG`
- `Probabilistic_model.PNG`
- `bezdekIris.data.txt`
- `fisher`
- `prob_gen_model.png`


Iris-setosa
Iris-versicolor and Iris-virginica
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.4
−0.3
−0.2
−0.1
0.0
0.1
0.2
Iris-setosa
Iris-versicolor and Iris-virginica
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.4
−0.3
−0.2
−0.1
0.0
0.1
0.2


3.0
2.5
2.0
1.5
1.0
0
10
20
30
40
50
Iris-setosa
Iris-versicolor
Iris-virginica

Particle filter

A basic particle filter tracking algorithm, using a uniformly distributed step as motion model, and the initial target colour as determinant feature for the weighting function. This requires an approximately uniformly coloured object, which moves at a speed no larger than stepsize per frame.

This implementation assumes that the video stream is a sequence of numpy arrays, an iterator pointing to such a sequence or a generator generating one. The particle filter itself is a generator to allow for operating on real-time video streams.

```
#!python
from numpy import *
from numpy.random import *

def resample(weights):
  n = len(weights)
  indices = []
  C = [0.] + [sum(weights[:i+1]) for i in range(n)]
  u0, j = random(), 0
  for u in [(u0+i)/n for i in range(n)]:
    while u > C[j]:
      j+=1
    indices.append(j-1)
  return indices

def particlefilter(sequence, pos, stepsize, n):
  seq = iter(sequence)
  x = ones((n, 2), int) * pos                   # Initial position
  f0 = seq.next()[tuple(pos)] * ones(n)         # Target colour moc
  yield pos, x, ones(n)/n                       # Return expected p
  for im in seq:
    x += uniform(-stepsize, stepsize, x.shape)  # Particle motion m
    x  = x.clip(zeros(2), array(im.shape)-1).astype(int) # Clip out
    f  = im[tuple(x.T)]                         # Measure particle
    w  = 1./(1. + (f0-f)**2)                    # Weight~ inverse c
    w /= sum(w)                                 # Normalize w
    yield sum(x.T*w, axis=1), x, w              # Return expected p
    if 1./sum(w**2) < n/2.:                     # If particle clouc
      x  = x[resample(w),:]                     # Resample particle
```

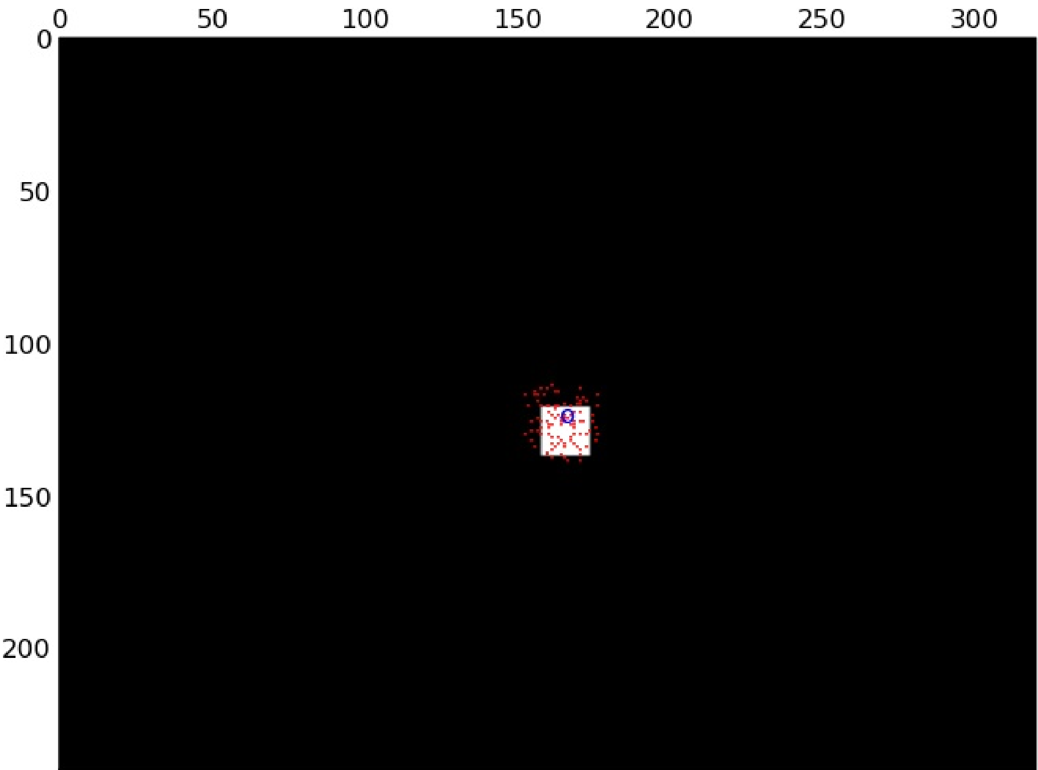
The following code shows the tracker operating on a test sequence featuring a moving square against a uniform background.

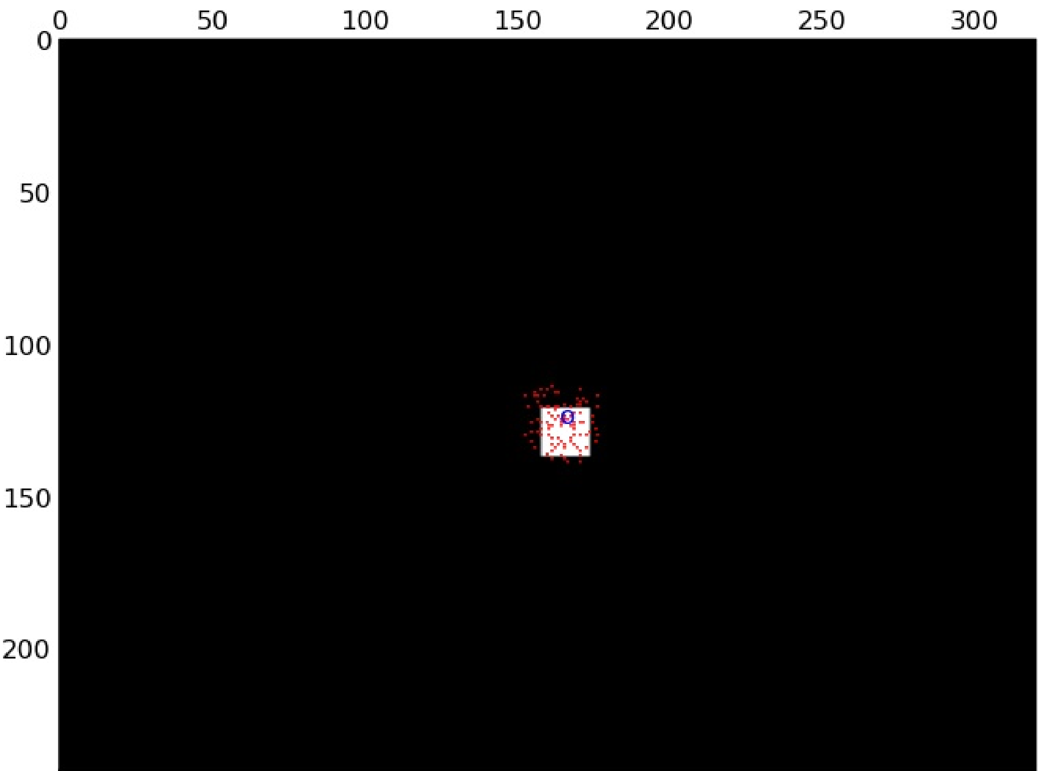
```
#!python
if __name__ == "__main__":
  from pylab import *
  from itertools import izip
  import time
  ion()
  seq = [ im for im in zeros((20,240,320), int)]      # Create an i
  x0 = array([120, 160])                              # Add a squar
  xs = vstack((arange(20)*3, arange(20)*2)).T + x0
  for t, x in enumerate(xs):
    xslice = slice(x[0]-8, x[0]+8)
    yslice = slice(x[1]-8, x[1]+8)
    seq[t][xslice, yslice] = 255

  for im, p in izip(seq, particlefilter(seq, x0, 8, 100)): # Track
    pos, xs, ws = p
    position_overlay = zeros_like(im)
    position_overlay[tuple(pos)] = 1
    particle_overlay = zeros_like(im)
    particle_overlay[tuple(xs.T)] = 1
    hold(True)
    draw()
    time.sleep(0.3)
    clf()                                           # Causes flicke
    imshow(im,cmap=cm.gray)                         # Plot the imag
    spy(position_overlay, marker='.', color='b')    # Plot the expe
    spy(particle_overlay, marker=',', color='r')    # Plot the part
  show()
```

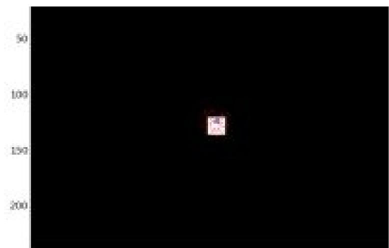
Attachments

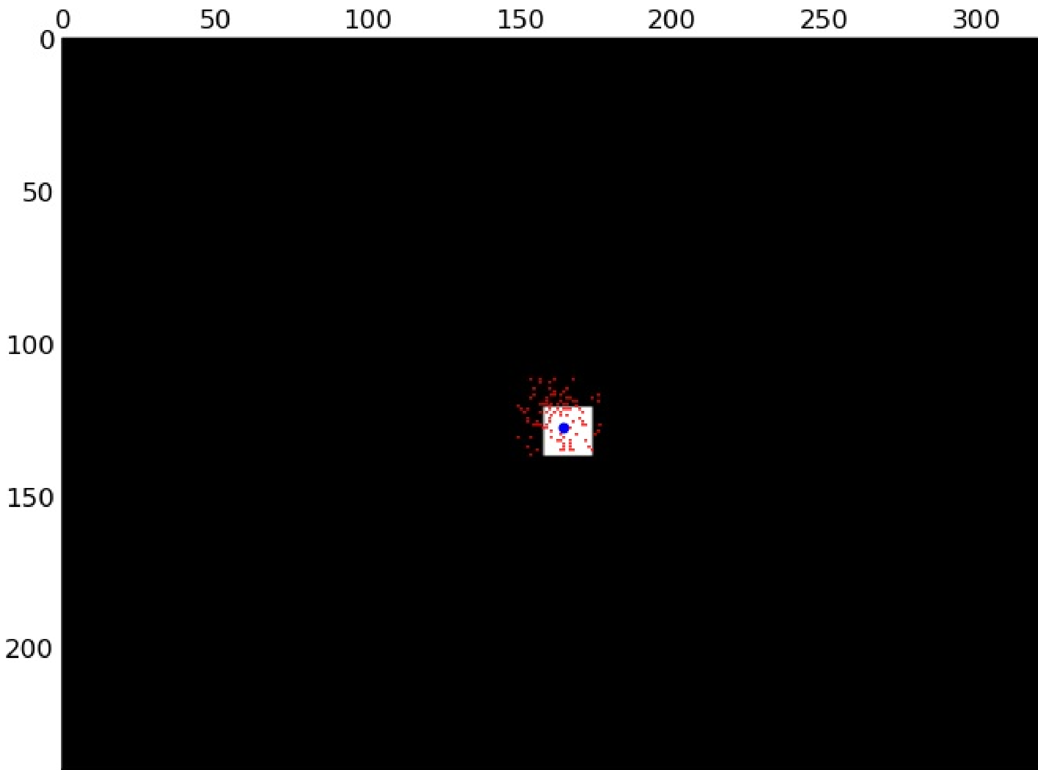
- `pftrack.jpg`
- `pftrack.png`
- `pftracking.jpg`
- `track.jpg`


0
50
100
150
200
250
300
0
50
100
150
200


0
50
100
150
200
250
300
0
50
100
150
200




50
100
150
200


0
50
100
150
200
250
300
0
50
100
150
200

Rebinning

Examples showing how to rebin data to produce a smaller or bigger array without (and with) using interpolation.

Example 1

Here we deal with the simplest case where any desired new shape is valid and no interpolation is done on the data to determine the new values. *First, floating slices objects are created for each dimension.* Second, the coordinates of the new bins are computed from the slices using mgrid. *Then, coordinates are transformed to integer indices.* And, finally, 'fancy indexing' is used to evaluate the original array at the desired indices.

```
def rebin( a, newshape ):
        '''Rebin an array to a new shape.
 '''
        assert len(a.shape) == len(newshape)

        slices = [ slice(0,old, float(old)/new) for old,new in zip(
        coordinates = mgrid[slices]
        indices = coordinates.astype('i')   #choose the biggest sma
        return a[tuple(indices)]
```

If we were only interested in reducing the sizes by some integer factor then we could use:

```
def rebin_factor( a, newshape ):
        '''Rebin an array to a new shape.
 newshape must be a factor of a.shape.
 '''
        assert len(a.shape) == len(newshape)
        assert not sometrue(mod( a.shape, newshape ))

        slices = [ slice(None,None, old/new) for old,new in zip(a.s
        return a[slices]
```

Example 2

Here is an other way to deal with the reducing case for ndarrays. This acts identically to IDL's rebin command where all values in the original array are summed and divided amongst the entries in the new array. As in IDL, the new shape must be a factor of the old one. The ugly 'evList trick' builds and executes a python command of the form

a.reshape(args[0],factor[0],).sum(1)/factor[0]
a.reshape(args[0],factor[0],args[1],factor[1],).sum(1).sum(2)/factor[0]/factor[1]

etc. This general form is extended to cover the number of required dimensions.

```
def rebin(a, *args):
    '''rebin ndarray data into a smaller ndarray of the same rank w
 are factors of the original dimensions. eg. An array with 6 columr
 can be reduced to have 6,3,2 or 1 columns and 4,2 or 1 rows.
 example usages:
 >>> a=rand(6,4); b=rebin(a,3,2)
 >>> a=rand(6); b=rebin(a,2)
 '''
    shape = a.shape
    lenShape = len(shape)
    factor = asarray(shape)/asarray(args)
    evList = ['a.reshape('] + \
             ['args[%d],factor[%d],'%(i,i) for i in range(lenShape)
             [')'] + ['.sum(%d)'%(i+1) for i in range(lenShape)] +
             ['/factor[%d]'%i for i in range(lenShape)]
    print ''.join(evList)
    return eval(''.join(evList))
```

The above code returns an array of the same type as the input array. If the input is an integer array, the output values will be rounded down. If you want a float array which correctly averages the input values without rounding, you can do the following instead.

a.reshape(args[0],factor[0],).mean(1)
a.reshape(args[0],factor[0],args[1],factor[1],).mean(1).mean(2)

```
def rebin(a, *args):
    shape = a.shape
    lenShape = len(shape)
    factor = asarray(shape)/asarray(args)
    evList = ['a.reshape('] + \
             ['args[%d],factor[%d],'%(i,i) for i in range(lenShape)
             [')'] + ['.mean(%d)'%(i+1) for i in range(lenShape)]
    print ''.join(evList)
    return eval(''.join(evList))
```

Some test cases:

```
# 2-D case
a=rand(6,4)
print a
b=rebin(a,6,4)
print b
b=rebin(a,6,2)
print b
b=rebin(a,3,2)
print b
b=rebin(a,1,1)

# 1-D case
print b
a=rand(4)
print a
b=rebin(a,4)
print b
b=rebin(a,2)
print b
b=rebin(a,1)
print b
```

Example 3

A python version of congrid, used in IDL, for resampling of data to arbitrary sizes, using a variety of nearest-neighbour and interpolation routines.

```
import numpy as n
import scipy.interpolate
import scipy.ndimage

def congrid(a, newdims, method='linear', centre=False, minusone=Fal
    '''Arbitrary resampling of source array to new dimension sizes
 Currently only supports maintaining the same number of dimensions
 To use 1-D arrays, first promote them to shape (x,1).

 Uses the same parameters and creates the same co-ordinate lookup p
 as IDL''s congrid routine, which apparently originally came from a
 routine of the same name.

 method:
 neighbour - closest value from original data
 nearest and linear - uses n x 1-D interpolations using
 scipy.interpolate.interp1d
 (see Numerical Recipes for validity of use of n 1-D interpolations
 spline - uses ndimage.map_coordinates
```

```
centre:
True - interpolation points are at the centres of the bins
False - points are at the front edge of the bin

minusone:
For example- inarray.shape = (i,j) & new dimensions = (x,y)
False - inarray is resampled by factors of (i/x) * (j/y)
True - inarray is resampled by(i-1)/(x-1) * (j-1)/(y-1)
This prevents extrapolation one element beyond bounds of input arr
'''
    if not a.dtype in [n.float64, n.float32]:
        a = n.cast[float](a)

    m1 = n.cast[int](minusone)
    ofs = n.cast[int](centre) * 0.5
    old = n.array( a.shape )
    ndims = len( a.shape )
    if len( newdims ) != ndims:
        print "[congrid] dimensions error. " \
              "This routine currently only support " \
              "rebinning to the same number of dimensions."
        return None
    newdims = n.asarray( newdims, dtype=float )
    dimlist = []

    if method == 'neighbour':
        for i in range( ndims ):
            base = n.indices(newdims)[i]
            dimlist.append( (old[i] - m1) / (newdims[i] - m1) \
                            * (base + ofs) - ofs )
        cd = n.array( dimlist ).round().astype(int)
        newa = a[list( cd )]
        return newa

    elif method in ['nearest','linear']:
        # calculate new dims
        for i in range( ndims ):
            base = n.arange( newdims[i] )
            dimlist.append( (old[i] - m1) / (newdims[i] - m1) \
                            * (base + ofs) - ofs )
        # specify old dims
        olddims = [n.arange(i, dtype = n.float) for i in list( a.sh

        # first interpolation - for ndims = any
        mint = scipy.interpolate.interp1d( olddims[-1], a, kind=met
        newa = mint( dimlist[-1] )

        trorder = [ndims - 1] + range( ndims - 1 )
        for i in range( ndims - 2, -1, -1 ):
            newa = newa.transpose( trorder )

            mint = scipy.interpolate.interp1d( olddims[i], newa, ki
            newa = mint( dimlist[i] )
```

```
        if ndims > 1:
            # need one more transpose to return to original dimensi
            newa = newa.transpose( trorder )

        return newa
    elif method in ['spline']:
        oslices = [ slice(0,j) for j in old ]
        oldcoords = n.ogrid[oslices]
        nslices = [ slice(0,j) for j in list(newdims) ]
        newcoords = n.mgrid[nslices]

        newcoords_dims = range(n.rank(newcoords))
        #make first index last
        newcoords_dims.append(newcoords_dims.pop(0))
        newcoords_tr = newcoords.transpose(newcoords_dims)
        # makes a view that affects newcoords

        newcoords_tr += ofs

        deltas = (n.asarray(old) - m1) / (newdims - m1)
        newcoords_tr *= deltas

        newcoords_tr -= ofs

        newa = scipy.ndimage.map_coordinates(a, newcoords)
        return newa
    else:
        print "Congrid error: Unrecognized interpolation type.\n", \
              "Currently only \'neighbour\', \'nearest\',\'linear\'
              "and \'spline\' are supported."
        return None
```

Savitzky Golay Filtering

The Savitzky Golay filter is a particular type of low-pass filter, well adapted for data smoothing. For further information see: http://www.wire.tu-bs.de/OLDWEB/mameyer/cmr/savgol.pdf (or http://www.dalkescientific.com/writings/NBN/data/savitzky_golay.py for a pre-numpy implementation).

Sample Code

```
#!python
def savitzky_golay(y, window_size, order, deriv=0, rate=1):
    r"""Smooth (and optionally differentiate) data with a Savitzky-
 The Savitzky-Golay filter removes high frequency noise from data.
 It has the advantage of preserving the original shape and
 features of the signal better than other types of filtering
 approaches, such as moving averages techniques.
 Parameters
 ----------
 y : array_like, shape (N,)
 the values of the time history of the signal.
 window_size : int
 the length of the window. Must be an odd integer number.
 order : int
 the order of the polynomial used in the filtering.
 Must be less then `window_size` - 1.
 deriv: int
 the order of the derivative to compute (default = 0 means only smo
 Returns
 -------
 ys : ndarray, shape (N)
 the smoothed signal (or it's n-th derivative).
 Notes
 -----
 The Savitzky-Golay is a type of low-pass filter, particularly
 suited for smoothing noisy data. The main idea behind this
 approach is to make for each point a least-square fit with a
 polynomial of high order over a odd-sized window centered at
 the point.
 Examples
 --------
 t = np.linspace(-4, 4, 500)
 y = np.exp( -t**2 ) + np.random.normal(0, 0.05, t.shape)
 ysg = savitzky_golay(y, window_size=31, order=4)
 import matplotlib.pyplot as plt
 plt.plot(t, y, label='Noisy signal')
 plt.plot(t, np.exp(-t**2), 'k', lw=1.5, label='Original signal')
```

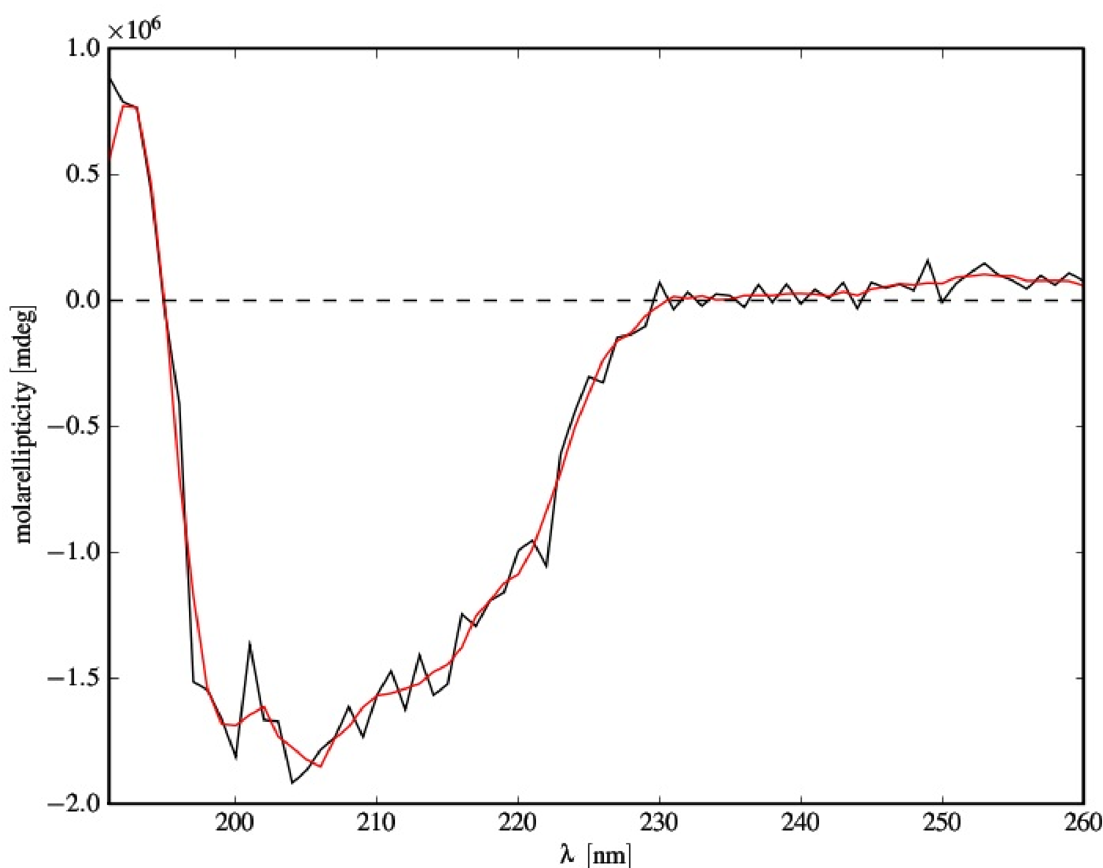
```
plt.plot(t, ysg, 'r', label='Filtered signal')
plt.legend()
plt.show()
References
----------
.. [1] A. Savitzky, M. J. E. Golay, Smoothing and Differentiation
Data by Simplified Least Squares Procedures. Analytical
Chemistry, 1964, 36 (8), pp 1627-1639.
.. [2] Numerical Recipes 3rd Edition: The Art of Scientific Comput
W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery
Cambridge University Press ISBN-13: 9780521880688
"""
    import numpy as np
    from math import factorial

    try:
        window_size = np.abs(np.int(window_size))
        order = np.abs(np.int(order))
    except ValueError, msg:
        raise ValueError("window_size and order have to be of type
    if window_size % 2 != 1 or window_size < 1:
        raise TypeError("window_size size must be a positive odd nu
    if window_size < order + 2:
        raise TypeError("window_size is too small for the polynomia
    order_range = range(order+1)
    half_window = (window_size -1) // 2
    # precompute coefficients
    b = np.mat([[k**i for i in order_range] for k in range(-half_wi
    m = np.linalg.pinv(b).A[deriv] * rate**deriv * factorial(deriv)
    # pad the signal at the extremes with
    # values taken from the signal itself
    firstvals = y[0] - np.abs( y[1:half_window+1][::-1] - y[0] )
    lastvals = y[-1] + np.abs(y[-half_window-1:-1][::-1] - y[-1])
    y = np.concatenate((firstvals, y, lastvals))
    return np.convolve( m[::-1], y, mode='valid')
```

Code explanation

In lines 61-62 the coefficients of the local least-square polynomial fit are pre-computed. These will be used later at line 68, where they will be correlated with the signal. To prevent spurious results at the extremes of the data, the signal is padded at both ends with its mirror image, (lines 65-67).

Figure



CD-spectrum of a protein. Black: raw data. Red: filter applied

A wrapper for cyclic voltammetry data

One of the most popular applications of S-G filter, apart from smoothing UV-VIS and IR spectra, is smoothing of curves obtained in electroanalytical experiments. In cyclic voltammetry, voltage (being the abcissa) changes like a triangle wave. And in the signal there are cusps at the turning points (at switching potentials) which should never be smoothed. In this case, Savitzky-Golay smoothing should be done piecewise, ie. separately on pieces monotonic in x:

```
#!python numbers=disable
def savitzky_golay_piecewise(xvals, data, kernel=11, order =4):
    turnpoint=0
    last=len(xvals)
    if xvals[1]>xvals[0] : #x is increasing?
        for i in range(1,last) : #yes
            if xvals[i]<xvals[i-1] : #search where x starts to fal
                turnpoint=i
                break
    else: #no, x is decreasing
        for i in range(1,last) : #search where it starts to rise
            if xvals[i]>xvals[i-1] :
                turnpoint=i
                break
    if turnpoint==0 : #no change in direction of x
        return savitzky_golay(data, kernel, order)
    else:
        #smooth the first piece
        firstpart=savitzky_golay(data[0:turnpoint],kernel,order)
        #recursively smooth the rest
        rest=savitzky_golay_piecewise(xvals[turnpoint:], data[turnp
        return numpy.concatenate((firstpart,rest))
```

Two dimensional data smoothing and least-square gradient estimate

Savitsky-Golay filters can also be used to smooth two dimensional data affected by noise. The algorithm is exactly the same as for the one dimensional case, only the math is a bit more tricky. The basic algorithm is as follow: 1. for each point of the two dimensional matrix extract a sub-matrix, centered at that point and with a size equal to an odd number “window_size”. 2. for this sub-matrix compute a least-square fit of a polynomial surface, defined as p(x,y) = a0 + a1*x* + *a2*y + a3*x*^2 + *a4*y^2 + a5*x*y + ... Note that x and y are equal to zero at the central point. 3. replace the initial central point with the value computed with the fit.

Note that because the fit coefficients are linear with respect to the data spacing, they can pre-computed for efficiency. Moreover, it is important to appropriately pad the borders of the data, with a mirror image of the data itself, so that the evaluation of the fit at the borders of the data can happen smoothly.

Here is the code for two dimensional filtering.

```
#!python numbers=enable
def sgolay2d ( z, window_size, order, derivative=None):
    """
 """
    # number of terms in the polynomial expression
```

```
    n_terms = ( order + 1 ) * ( order + 2)  / 2.0

    if  window_size % 2 == 0:
        raise ValueError('window_size must be odd')

    if window_size**2 < n_terms:
        raise ValueError('order is too high for the window size')

    half_size = window_size // 2

    # exponents of the polynomial.
    # p(x,y) = a0 + a1*x + a2*y + a3*x^2 + a4*y^2 + a5*x*y + ...
    # this line gives a list of two item tuple. Each tuple contains
    # the exponents of the k-th term. First element of tuple is for
    # second element for y.
    # Ex. exps = [(0,0), (1,0), (0,1), (2,0), (1,1), (0,2), ...]
    exps = [ (k-n, n) for k in range(order+1) for n in range(k+1) ]

    # coordinates of points
    ind = np.arange(-half_size, half_size+1, dtype=np.float64)
    dx = np.repeat( ind, window_size )
    dy = np.tile( ind, [window_size, 1]).reshape(window_size**2, )

    # build matrix of system of equation
    A = np.empty( (window_size**2, len(exps)) )
    for i, exp in enumerate( exps ):
        A[:,i] = (dx**exp[0]) * (dy**exp[1])

    # pad input array with appropriate values at the four borders
    new_shape = z.shape[0] + 2*half_size, z.shape[1] + 2*half_size
    Z = np.zeros( (new_shape) )
    # top band
    band = z[0, :]
    Z[:half_size, half_size:-half_size] =  band -  np.abs( np.flipu
    # bottom band
    band = z[-1, :]
    Z[-half_size:, half_size:-half_size] = band  + np.abs( np.flipu
    # left band
    band = np.tile( z[:,0].reshape(-1,1), [1,half_size])
    Z[half_size:-half_size, :half_size] = band - np.abs( np.fliplr(
    # right band
    band = np.tile( z[:,-1].reshape(-1,1), [1,half_size] )
    Z[half_size:-half_size, -half_size:] =  band + np.abs( np.flipl
    # central band
    Z[half_size:-half_size, half_size:-half_size] = z

    # top left corner
    band = z[0,0]
    Z[:half_size,:half_size] = band - np.abs( np.flipud(np.fliplr(z
    # bottom right corner
    band = z[-1,-1]
    Z[-half_size:,-half_size:] = band + np.abs( np.flipud(np.fliplr
```

```
    # top right corner
    band = Z[half_size,-half_size:]
    Z[:half_size,-half_size:] = band - np.abs( np.flipud(Z[half_siz
    # bottom left corner
    band = Z[-half_size:,half_size].reshape(-1,1)
    Z[-half_size:,:half_size] = band - np.abs( np.fliplr(Z[-half_si

    # solve system and convolve
    if derivative == None:
        m = np.linalg.pinv(A)[0].reshape((window_size, -1))
        return scipy.signal.fftconvolve(Z, m, mode='valid')
    elif derivative == 'col':
        c = np.linalg.pinv(A)[1].reshape((window_size, -1))
        return scipy.signal.fftconvolve(Z, -c, mode='valid')
    elif derivative == 'row':
        r = np.linalg.pinv(A)[2].reshape((window_size, -1))
        return scipy.signal.fftconvolve(Z, -r, mode='valid')
    elif derivative == 'both':
        c = np.linalg.pinv(A)[1].reshape((window_size, -1))
        r = np.linalg.pinv(A)[2].reshape((window_size, -1))
        return scipy.signal.fftconvolve(Z, -r, mode='valid'), scipy
```

Here is a demo

```
#!python number=enable

# create some sample twoD data
x = np.linspace(-3,3,100)
y = np.linspace(-3,3,100)
X, Y = np.meshgrid(x,y)
Z = np.exp( -(X**2+Y**2))

# add noise
Zn = Z + np.random.normal( 0, 0.2, Z.shape )

# filter it
Zf = sgolay2d( Zn, window_size=29, order=4)

# do some plotting
matshow(Z)
matshow(Zn)
matshow(Zf)
```

attachment:Original.pdf Original data attachment:Original+noise.pdf Original data + noise attachment:Original+noise+filtered.pdf (Original data + noise) filtered

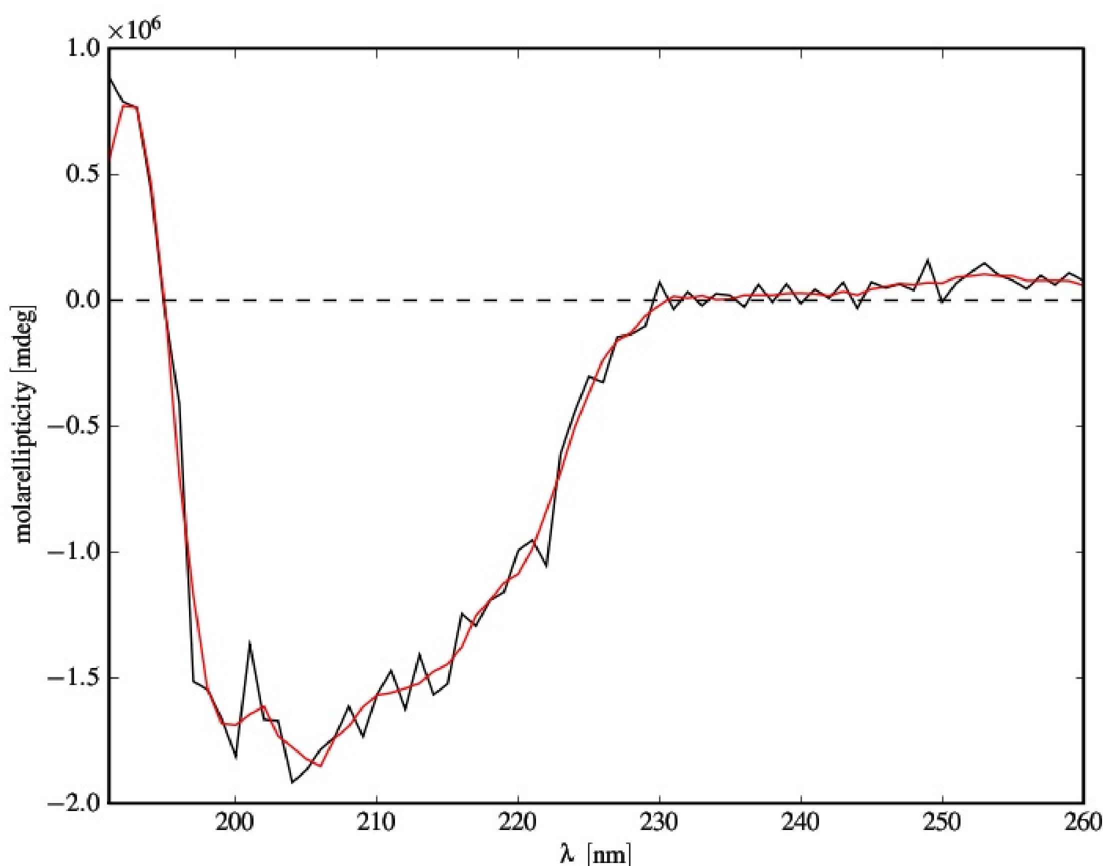
Gradient of a two-dimensional function

Since we have computed the best fitting interpolating polynomial surface it is easy to compute its gradient. This method of computing the gradient of a two dimensional function is quite robust, and partially hides the noise in the data, which strongly affects the differentiation operation. The maximum order of the derivative that can be computed obviously depends on the order of the polynomial used in the fitting.

The code provided above have an option `derivative`, which as of now allows to compute the first derivative of the 2D data. It can be "row"or "column", indicating the direction of the derivative, or "both", which returns the gradient.

Attachments

- `Original+noise+filtered.pdf`
- `Original+noise.pdf`
- `Original.pdf`
- `cd_spec.png`



Smoothing of a 1D signal

This method is based on the convolution of a scaled window with the signal. The signal is prepared by introducing reflected window-length copies of the signal at both ends so that boundary effect are minimized in the beginning and end part of the output signal.

Code

```
import numpy

def smooth(x,window_len=11,window='hanning'):
    """smooth the data using a window with requested size.

    This method is based on the convolution of a scaled window with th
    The signal is prepared by introducing reflected copies of the sign
    (with the window size) in both ends so that transient parts are mi
    in the begining and end part of the output signal.

    input:
    x: the input signal
    window_len: the dimension of the smoothing window; should be an od
    window: the type of window from 'flat', 'hanning', 'hamming', 'bar
    flat window will produce a moving average smoothing.

    output:
    the smoothed signal

    example:

    t=linspace(-2,2,0.1)
    x=sin(t)+randn(len(t))*0.1
    y=smooth(x)

    see also:

    numpy.hanning, numpy.hamming, numpy.bartlett, numpy.blackman, nump
    scipy.signal.lfilter

    TODO: the window parameter could be the window itself if an array
    NOTE: length(output) != length(input), to correct this: return y[(
    """

    if x.ndim != 1:
        raise ValueError, "smooth only accepts 1 dimension arrays."

    if x.size < window_len:
```

```
        raise ValueError, "Input vector needs to be bigger than win

    if window_len<3:
        return x

    if not window in ['flat', 'hanning', 'hamming', 'bartlett', 'bl
        raise ValueError, "Window is on of 'flat', 'hanning', 'hamm

    s=numpy.r_[x[window_len-1:0:-1],x,x[-1:-window_len:-1]]
    #print(len(s))
    if window == 'flat': #moving average
        w=numpy.ones(window_len,'d')
    else:
        w=eval('numpy.'+window+'(window_len)')

    y=numpy.convolve(w/w.sum(),s,mode='valid')
    return y

from numpy import *
from pylab import *

def smooth_demo():

    t=linspace(-4,4,100)
    x=sin(t)
    xn=x+randn(len(t))*0.1
    y=smooth(x)

    ws=31

    subplot(211)
    plot(ones(ws))

    windows=['flat', 'hanning', 'hamming', 'bartlett', 'blackman']

    hold(True)
    for w in windows[1:]:
        eval('plot('+w+'(ws) )')

    axis([0,30,0,1.1])

    legend(windows)
    title("The smoothing windows")
    subplot(212)
    plot(x)
    plot(xn)
    for w in windows:
        plot(smooth(xn,10,w))
    l=['original signal', 'signal with noise']
    l.extend(windows)

    legend(l)
    title("Smoothing a noisy signal")
```

```
    show()

if __name__=='__main__':
    smooth_demo()
```

Figure

[](files/attachments/SignalSmooth/smoothsignal.jpg

Solving large Markov Chains

This page shows how to compute the stationary distribution pi of a large Markov chain. The example is a tandem of two M/M/1 queues. Generally the transition matrix P of the Markov chain is sparse, so that we can either use scipy.sparse or Pysparse. Here we demonstrate how to use both of these tools.

Power Method

In this section we find pi by means of iterative methods called the Power method. More specifically, given a (stochastic) transition matrix P, and an initial vector pi*0, compute iteratively pi_n = pi*{n-1} P until the distance (in some norm) between pi*n and pi*{n-1} is small enough.

Fist we build the generator matrix Q for the related Markov chain. Then we turn Q into a transition matrix P by the method of uniformization, that is, we define P as I - Q/l, where I is the identity matrix (of the same size as Q) and l is the smallest element on the diagonal of Q. Once we have P, we approximate pi (the left eigenvector of P that satisfies pi = pi P) by the iterates pi_n = pi_0 P^n, for some initial vector pi_0.

More details of the above approach can be found in (more or less) any book on probability and Markov Chains. A fine example, with many nice examples and attention to the numerical solution of Markov chains, is `Queueing networks and Markov Chains' by G. Bolch et al., John Wiley, 2nd edition, 2006.

You can get the source code for this tutorial here: `tandemqueue.py`

```
#!/usr/bin/env python

labda, mu1, mu2 = 1., 1.01, 1.001
N1, N2 = 50, 50
size = N1*N2

from numpy import ones, zeros, empty
import scipy.sparse as sp
import  pysparse
from pylab import matshow, savefig
from scipy.linalg import norm
import time
```

This simple function converts the state (i,j), which represents that the first queue contains i jobs and the second queue j jobs, to a more suitable form to define a transition matrix.

```
def state(i,j):
    return j*N1 + i
```

Build the off-diagonal elements of the generator matrix Q.

```
def fillOffDiagonal(Q):
    # labda
    for i in range(0,N1-1):
        for j in range(0,N2):
            Q[(state(i,j),state(i+1,j))]= labda
    # mu2
    for i in range(0,N1):
        for j in range(1,N2):
            Q[(state(i,j),state(i,j-1))]= mu2
    # mu1
    for i in range(1,N1):
        for j in range(0,N2-1):
            Q[(state(i,j),state(i-1,j+1))]= mu1
    print "ready filling"
```

In this function we use scipy.sparse

```
def computePiMethod1():
    e0 = time.time()
    Q = sp.dok_matrix((size,size))
    fillOffDiagonal(Q)
    # Set the diagonal of Q such that the row sums are zero
    Q.setdiag( -Q*ones(size) )
    # Compute a suitable stochastic matrix by means of uniformizati
    l = min(Q.values())*1.001  # avoid periodicity, see the book of
    P = sp.speye(size, size) - Q/l
    # compute Pi
    P =  P.tocsr()
    pi = zeros(size);  pi1 = zeros(size)
    pi[0] = 1;
    n = norm(pi - pi1,1); i = 0;
    while n > 1e-3 and i < 1e5:
        pi1 = pi*P
        pi = pi1*P   # avoid copying pi1 to pi
        n = norm(pi - pi1,1); i += 1
    print "Method 1: ", time.time() - e0, i
    return pi
```

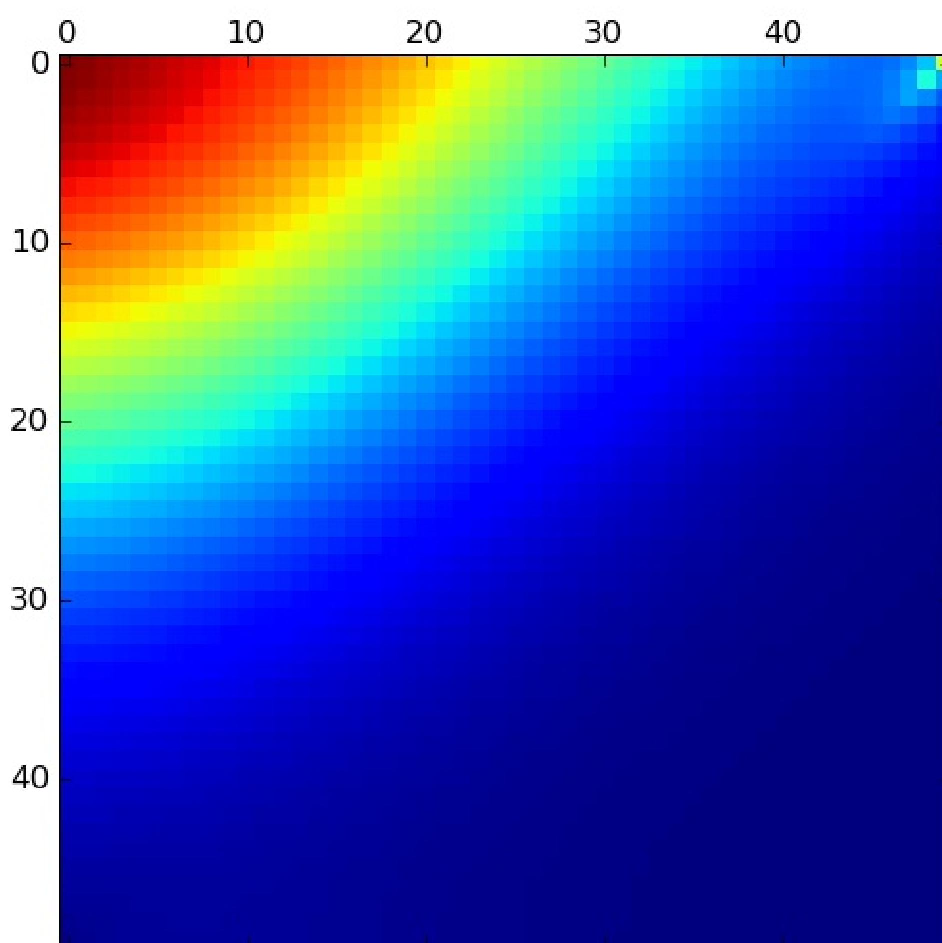
Now use Pysparse.

```
def computePiMethod2():
    e0 = time.time()
    Q = pysparse.spmatrix.ll_mat(size,size)
    fillOffDiagonal(Q)
    # fill diagonal
    x =  empty(size)
    Q.matvec(ones(size),x)
    Q.put(-x)
    # uniformize
    l = min(Q.values())*1.001
    P = pysparse.spmatrix.ll_mat(size,size)
    P.put(ones(size))
    P.shift(-1./l, Q)
    # Compute pi
    P = P.to_csr()
    pi = zeros(size);  pi1 = zeros(size)
    pi[0] = 1;
    n = norm(pi - pi1,1); i = 0;
    while n > 1e-3 and i < 1e5:
        P.matvec_transp(pi,pi1)
        P.matvec_transp(pi1,pi)
        n = norm(pi - pi1,1); i += 1
    print "Method 2: ", time.time() - e0, i
    return pi
```

Output the results.

```
def plotPi(pi):
    pi = pi.reshape(N2,N1)
    matshow(pi)
    savefig("pi.eps")
if __name__ == "__main__":
    pi = computePiMethod1()
    pi = computePiMethod2()
    plotPi(pi)
```

Here is the result:

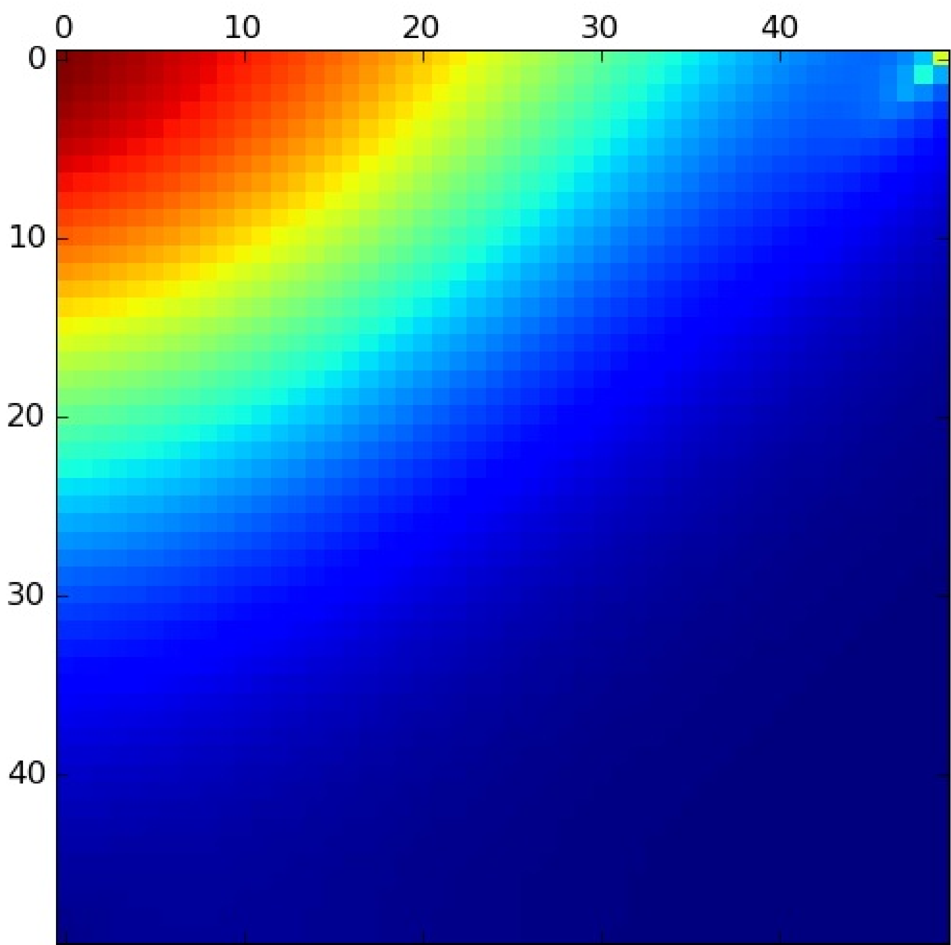


Improvements of this Tutorial

Include other methods such as Jacobi's method or Gauss Seidel.

Attachments

- `pi.png`
- `tandemqueue.py`


0
10
20
30
40
0
10
20
30
40

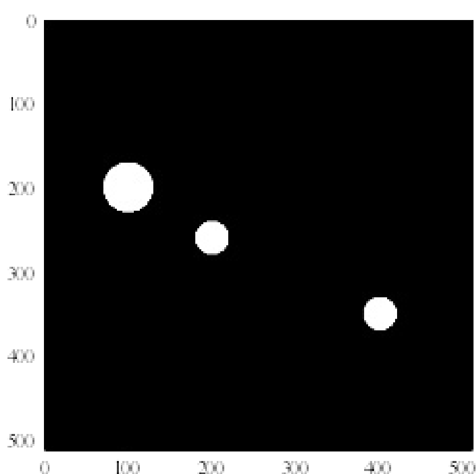
Watershed

The watershed algorithm (see 1)) is used to split an image into distinct components.

Suppose that we have the following image, composed of three whites disks (pixels of value 1) and a black background (pixels of value 0). We want to obtain a new array where each pixel is labeled with the index of the component to which it belongs, that is a segmentation of the orginal array, as shown in the image below. We will use the watershed algorithm provided by scipy.ndimage, scipy.ndimage.watershed_ift.

```
# Create the initial black and white image
import numpy as np
from scipy import ndimage
import matplotlib.pyplot as plt
a = np.zeros((512, 512)).astype(np.uint8) #unsigned integer type ne
y, x = np.ogrid[0:512, 0:512]
m1 = ((y-200)**2 + (x-100)**2 < 30**2)
m2 = ((y-350)**2 + (x-400)**2 < 20**2)
m3 = ((y-260)**2 + (x-200)**2 < 20**2)
a[m1+m2+m3]=1
plt.imshow(a, cmap='gray')# left plot in the image above
```

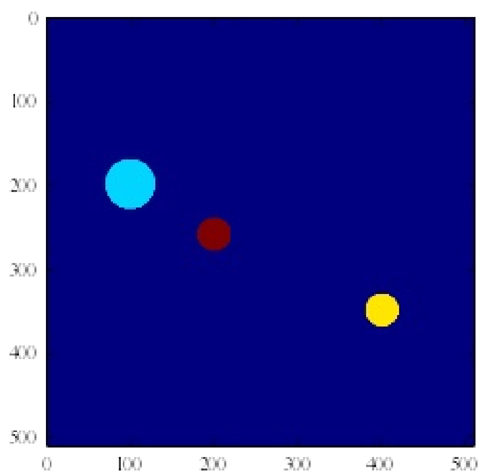
```
<matplotlib.image.AxesImage at 0x7ff9c8348690>
```



The watershed algorithm relies on the *flooding* of different basins, so we need to put markers in the image to initiate the flooding. If one knows approximately where the objects are, and there are only a few objects, it is possible to set the markers by hand

```
markers = np.zeros_like(a).astype(np.int16)
markers[0, 0] = 1
markers[200, 100] = 2
markers[350, 400] = 3
markers[260, 200] = 4
res1 = ndimage.watershed_ift(a.astype(np.uint8), markers)
plt.imshow(res1, cmap='jet') # central plot in the image above
```

```
<matplotlib.image.AxesImage at 0x7ff9c827a550>
```



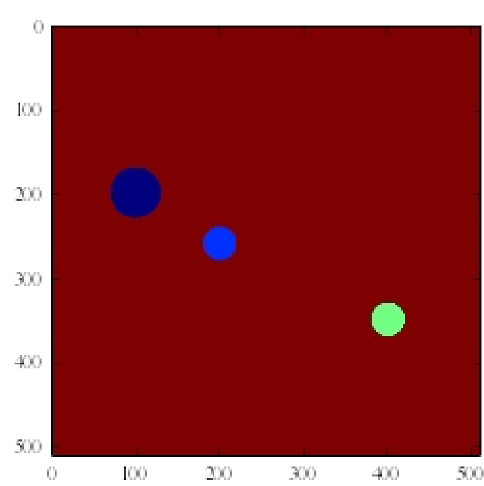

```
np.unique(res1) # pixels are tagged according to the object they be
```

```
array([1, 2, 3, 4], dtype=int16)
```

If you don’t know where to put the markers, and you know the minimal size of the objects, you can spread a lot of markers on a grid finer than the objects.

```
xm, ym = np.ogrid[0:512:10, 0:512:10]
markers = np.zeros_like(a).astype(np.int16)
markers[xm, ym]= np.arange(xm.size*ym.size).reshape((xm.size,ym.si
res2 = ndimage.watershed_ift(a.astype(np.uint8), markers)
res2[xm, ym] = res2[xm-1, ym-1] # remove the isolate seeds
plt.imshow(res2, cmap='jet')
```

```
<matplotlib.image.AxesImage at 0x7ff9c81a3a10>
```


0
100
200
300
400
500
0
100
200
300
400
500

Numpy & Scipy / Root finding

Function intersections

Find the points at which two given functions intersect

Consider the example of finding the intersection of a polynomial and a line:

$y_1=x_1^2$

$y_2=x_2+1$

```
from scipy.optimize import fsolve

import numpy as np

def f(xy):
   x, y = xy
   z = np.array([y - x**2, y - x - 1.0])
   return z

fsolve(f, [1.0, 2.0])
```

```
array([ 1.61803399,  2.61803399])
```

See also: http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fsolve.html#scipy.optimize.fsolve

Spherical Bessel Zeros

It may be useful to find out the zeros of the spherical Bessel functions, for instance, if you want to compute the eigenfrequencies of a spherical electromagnetic cavity (in this case, you'll need also the zeros of the derivative of (r*Jn(r))).

The problem is that you have to work out the ranges where you are supposed to find the zeros.

Happily, the range of a given zero of the n'th spherical Bessel functions can be computed from the zeros of the (n-1)'th spherical Bessel function.

Thus, the approach proposed here is recursive, knowing that the spherical Bessel function of order 0 is equal to sin(r)/r, whose zeros are well known.

This approach is obviously not efficient at all, but it works ;-).

A sample example is shown, for the 10 first zeros of the spherical Bessel function of order 5 (and the derivative of (r*J5(r))), using [:Cookbook/Matplotlib: matplotlib].

```
#! /usr/bin/env python

### recursive method: computes zeros ranges of Jn(r,n) from zeros o
### (also for zeros of (rJn(r,n))')
### pros : you are certain to find the right zeros values;
### cons : all zeros of the n-1 previous Jn have to be computed;
### note : Jn(r,0) = sin(r)/r

from scipy import arange, pi, sqrt, zeros
from scipy.special import jv, jvp
from scipy.optimize import brentq
from sys import argv
from pylab import *

def Jn(r,n):
  return (sqrt(pi/(2*r))*jv(n+0.5,r))
def Jn_zeros(n,nt):
  zerosj = zeros((n+1, nt), dtype=Float32)
  zerosj[0] = arange(1,nt+1)*pi
  points = arange(1,nt+n+1)*pi
  racines = zeros(nt+n, dtype=Float32)
  for i in range(1,n+1):
    for j in range(nt+n-i):
      foo = brentq(Jn, points[j], points[j+1], (i,))
      racines[j] = foo
    points = racines
    zerosj[i][:nt] = racines[:nt]
  return (zerosj)
```

```
def rJnp(r,n):
  return (0.5*sqrt(pi/(2*r))*jv(n+0.5,r) + sqrt(pi*r/2)*jvp(n+0.5,r
def rJnp_zeros(n,nt):
  zerosj = zeros((n+1, nt), dtype=Float32)
  zerosj[0] = (2.*arange(1,nt+1)-1)*pi/2
  points = (2.*arange(1,nt+n+1)-1)*pi/2
  racines = zeros(nt+n, dtype=Float32)
  for i in range(1,n+1):
    for j in range(nt+n-i):
      foo = brentq(rJnp, points[j], points[j+1], (i,))
      racines[j] = foo
    points = racines
    zerosj[i][:nt] = racines[:nt]
  return (zerosj)

n = int(argv[1])  # n'th spherical bessel function
nt = int(argv[2]) # number of zeros to be computed

dr = 0.01
eps = dr/1000

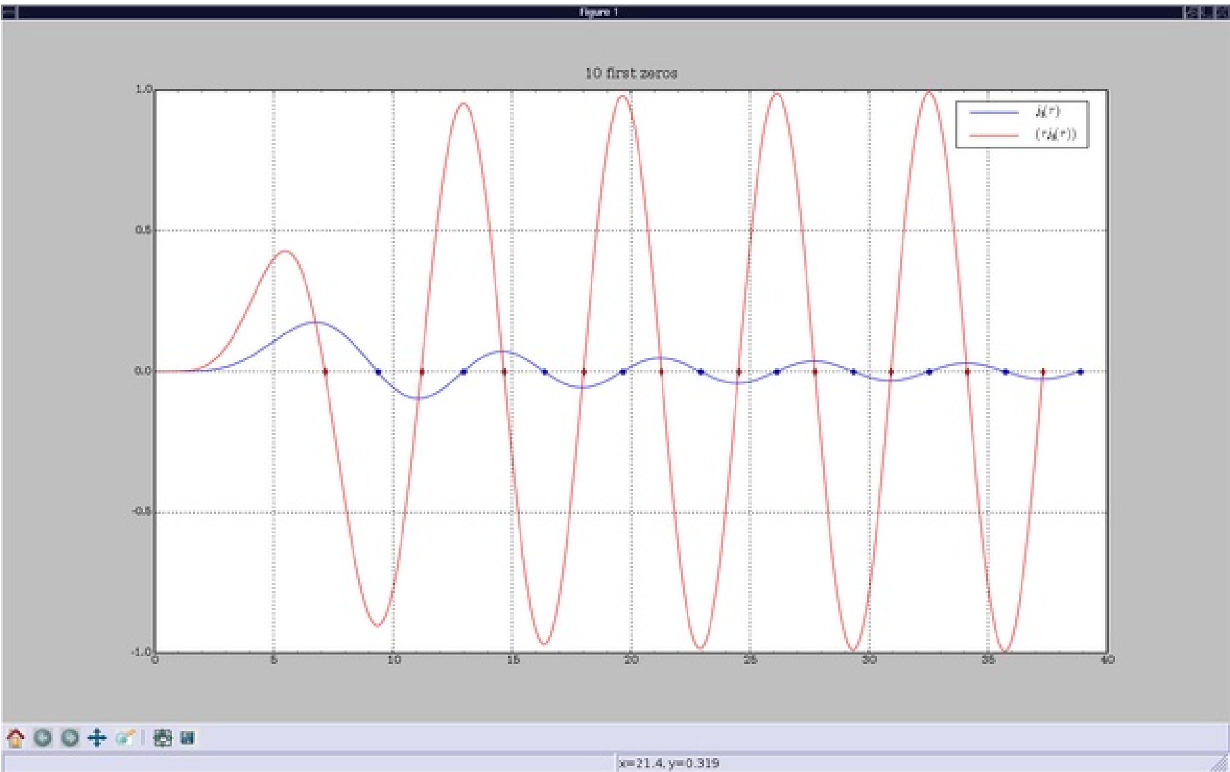
jnz = Jn_zeros(n,nt)[n]
r1 = arange(eps,jnz[len(jnz)-1],dr)
jnzp = rJnp_zeros(n,nt)[n]
r2 = arange(eps,jnzp[len(jnzp)-1],dr)

grid(True)
plot(r1,Jn(r1,n),'b', r2,rJnp(r2,n),'r')
title((str(nt)+' first zeros'))
legend((r'$j_{'+str(n)+'}(r)$', r'$(rj_{'+str(n)+'}(r))\'$'))
plot(jnz,zeros(len(jnz)),'bo', jnzp,zeros(len(jnzp)),'rd')
gca().xaxis.set_minor_locator(MultipleLocator(1))
# gca().xaxis.set_minor_formatter(FormatStrFormatter('%d'))
show()
```

```
bessph_zeros_rec 5 10
```

Attachments

- `snapshot.png`


figure 1
10 first zeros
1.0
0.5
0.0
-0.5
-1.0
0
5
10
15
20
25
30
35
40
x=21.4, y=0.319

Numpy & Scipy / Tips and tricks

Addressing Array Columns by Name

There are two very closely related ways to access array columns by name: recarrays and structured arrays. Structured arrays are just ndarrays with a complicated data type:

```
#!python numbers=disable
In [1]: from numpy import *
In [2]: ones(3, dtype=dtype([('foo', int), ('bar', float)]))
Out[2]:
array([(1, 1.0), (1, 1.0), (1, 1.0)],
      dtype=[('foo', '<i4'), ('bar', '<f8')])
In [3]: r = _
In [4]: r['foo']
Out[4]: array([1, 1, 1])
```

recarray is a subclass of ndarray that just adds attribute access to structured arrays:

```
#!python numbers=disable
In [10]: r2 = r.view(recarray)
In [11]: r2
Out[11]:
recarray([(1, 1.0), (1, 1.0), (1, 1.0)],
      dtype=[('foo', '<i4'), ('bar', '<f8')])
In [12]: r2.foo
Out[12]: array([1, 1, 1])
```

One downside of recarrays is that the attribute access feature slows down all field accesses, even the r['foo'] form, because it sticks a bunch of pure Python code in the middle. Much code won't notice this, but if you end up having to iterate over an array of records, this will be a hotspot for you.

Structured arrays are sometimes confusingly called record arrays.

```
. - lightly adapted from a Robert Kern post of Thu, 26 Jun 2008 15:2
```

Building arrays

This is a brief introduction to array objects, their declaration and use in scipy. A comprehensive list of examples of Numpy functions for arrays can be found at Numpy Example List With Doc

Basics

Numerical arrays are not yet defined in the standard python language. To load the array object and its methods into the namespace, the numpy package must be imported:

```
from numpy import *
```

Arrays can be created from the usual python lists and tuples using the array function. For example,

```
a = array([1,2,3])
```

returns a one dimensional array of integers. The array instance has a large set of methods and properties attached to it. For example, is the dimension of the array. In this case, it would simply be .

One big difference between array objects and python's sequences object is the definition of the mathematical operators. Whereas the addition of two lists concatenates those list, the addition of two arrays adds the arrays element-wise. For example :

```
b = array((10,11,12))
a + b
```

```
array([11, 13, 15])
```

Subtraction, multiplication and division are defined similarly.

A common gotcha for beginners is the type definition of arrays. Unless otherwise instructed, the array construct uses the type of its argument. Since was created from a list of integers, it is defined as an integer array, more precisely :

```
a.dtype
```

```
dtype('int64')
```

Accordingly, mathematical operations such as division will operate as usual in python, that is, will return an integer answer :

```
a/3
```

```
array([0, 0, 1])
```

To obtain the expected answer, one solution is to force the casting of integers into real numbers by dividing by a real number . A more careful approach is to define the type at initialization time :

```
a = array([1,2,3], dtype=float)
```

Another way to cast is by using Numpy's built-in cast functions astype and cast. These allow you to change the type of data you're working with:

```
a = array([1,2,3], dtype=int)
b = a.astype('float')
```

The elements of an array are accessed using the bracket notation where is an integer index starting at 0. Sub-arrays can be accessed by using general indexes of the form `start:stop:step` . `a[start:stop:step]` will return a reference to a sub-array of array `a` starting with (including) the element at index `start` going up to (but not including) the element at index stop in steps of *step*. e.g.:

```
data = array([0.5, 1.2, 2.2, 3.4, 3.5, 3.4, 3.4, 3.4], float)
t = arange(len(data), dtype='float') * 2*pi/(len(data)-1)
t[:]              # get all t-values
```

```
array([ 0.        ,  0.8975979 ,  1.7951958 ,  2.6927937 ,  3.590
        4.48798951,  5.38558741,  6.28318531])
```

```
t[2:4]             # get sub-array with the elements at the indexes
```

```
array([ 1.7951958,  2.6927937])
```

```
t[slice(2,4)]      # the same using slice
```

```
array([ 1.7951958,  2.6927937])
```

```
t[0:6:2]           # every even-indexed value up to but excluding 6
```

```
array([ 0.        ,  1.7951958,  3.5903916])
```

Furthermore, there is the possibility to access array-elements using bool-arrays. The bool-array has the indexes of elements which are to be accessed set to *True.*

```
i = array(len(t)*[False], bool)        # create an bool-array for i
i[2] = True; i[4] = True; i[6] = True  # we want elements with inde
t[i]
```

```
array([ 1.7951958 ,  3.5903916 ,  5.38558741])
```

We can use this syntax to make slightly more elaborate constructs. Consider the `data[:]` and `t[:]` arrays defined before. Suppose we want to get the four (`t[i]/data[i]`)-pairs with the four `t[i]-values` being closest to a point `p=1.8` . We could proceed as follows:

```
p=1.8                               # set our point
abs(t-p)                            # how much do the t[:]-values dif
```

```
array([ 1.8       ,  0.9024021 ,  0.0048042 ,  0.8927937 ,  1.790
        2.68798951,  3.58558741,  4.48318531])
```

```
dt_m = sort(abs(t-p))[3]        # how large is the 4-th largest ab
                                # t[:]-values and p
```

```
abs(t-p) <= dt_m                # where are the four elements of t
```

```
 array([False,  True,  True,  True,  True, False, False, False], dt
```

```
y_p = data[abs(t-p) <= dt_m]    # construct the sub-arrays; (1) ge
t_p = t[abs(t-p) <= dt_m]       # (2) get the data t[:]-values cor
y_p
```

```
 array([ 1.2,  2.2,  3.4,  3.5])
```

```
t_p
```

```
 array([ 0.8975979,  1.7951958,  2.6927937,  3.5903916])
```

It has to be kept in mind that slicing returns a reference to the data. As a consequence, changes in the returned sub-array cause changes in the original array and vice versa. If one wants to copy only the values one can use the copy()-method of the matrix object. For example:

```
# first lets define a 2-d matrix
A = array([[0,   1,  2,  3],   # initialize 2-d array
           [4,   5,  6,  7],
           [8,   9, 10, 11],
           [12, 13, 14, 15]])
A
```

```
 array([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]])
```

```
b=A[1:3,0:2]                    # let's get a sub-matrix containing
                                # rows 1,2 and columns 0,1
                                # !attention! this assigns to b a r
                                # sub-matrix of A
b
```

```
array([[4, 5],
       [8, 9]])
```

```
c=A[1:3,0:2].copy()             # copy the entries
c
```

```
array([[4, 5],
       [8, 9]])
```

```
A[1:3,0:2] = 42                 # we can also assign by slicing (th
b                               # b also affected (only a reference
```

```
array([[42, 42],
       [42, 42]])
```

```
c                               # still the same (deep copy)
```

```
array([[4, 5],
       [8, 9]])
```

Matrix dot product

The next example creates two matrices: `a` and `b`, and computes the dot product `axb` (in other words, the standard matrix product)

```
a = array([[1,2], [2,3]])
b = array([[7,1], [0,1]])
dot(a, b)
```

```
array([[ 7,  3],
       [14,  5]])
```

Automatic array creation

Scipy (via Numpy) provides numerous ways to create arrays automatically. For example, to create a vector of evenly spaced numbers, the linspace function can be called. This is often useful to compute the result of a function on some domain. For example, to compute the value of the function on one period, we would define a vector going from 0 to 2 pi and compute the value of the function for all values in this vector :

```
x = linspace(0, 2*pi, 100)
y = sin(x)
```

The same can be done on a N dimensional grid using the class and some of its object creation methods and . For example,

```
x, y = mgrid[0:10:.1, 0:10:.2]
```

returns two matrices, x and y, whose elements range from 0 to 10 (non-inclusively) in .1 and .2 increments respectively. These matrices can be used to compute the value of a function at the points (x_i, y_i) defined by those grids :

```
z = (x+y)**2
```

The ogrid object has the exact same behavior, but instead of storing an N-D matrix into memory, it stores only the 1-D vector that defines it. For large matrices, this can lead to significant economy of memory space.

Other useful functions to create matrices are and who initialize arrays full of zeros and ones. Note that those will be float arrays by default. This may lead to curious behaviour for the unawares. For example, let’s initialize a matrix with zeros, and then place values in it element by element.

```
mz = zeros((2, 2), dtype=int)
mz[0, 0] = .5**2
mz[1, 1] = 1.6**2
```

In this example, we are trying to store floating point numbers in an integer array. Thus, the numbers are then recast to integers, so that if we print the matrix, we obtain :

```
mz
```

```
array([[0, 0],
       [0, 2]])
```

To create real number arrays, one simply need to state the type explicitly in the call to the function :

```
mz = zeros((2, 2), dtype=float)
```

Repeating array segments

The ndarray.repeat() method returns a new array with dimensions repeated from the old one.

```
a = array([[0, 1],
...        [2, 3]])
a.repeat(2, axis=0) # repeats each row twice in succession
```

```
array([[0, 1],
       [0, 1],
       [2, 3],
       [2, 3]])
```

```
a.repeat(3, axis=1) # repeats each column 3 times in succession
```

```
array([[0, 0, 0, 1, 1, 1],
       [2, 2, 2, 3, 3, 3]])
```

```
a.repeat(2, axis=None) # flattens (ravels), then repeats each eleme
```

```
array([0, 0, 1, 1, 2, 2, 3, 3])
```

These can be combined to do some useful things, like enlarging image data stored in a 2D array:

```
def enlarge(a, x=2, y=None):
    """Enlarges 2D image array a using simple pixel repetition in b
 Enlarges by factor x horizontally and factor y vertically.
 If y is left as None, uses factor x for both dimensions."""
    a = asarray(a)
    assert a.ndim == 2
    if y == None:
        y = x
    for factor in (x, y):
        assert factor.__class__ == int
        assert factor > 0
    return a.repeat(y, axis=0).repeat(x, axis=1)

enlarge(a, x=2, y=2)
```

```
 array([[0, 0, 1, 1],
        [0, 0, 1, 1],
        [2, 2, 3, 3],
        [2, 2, 3, 3]])
```

Convolution-like operations

Users frequently want to break an array up into overlapping chunks, then apply the same operation to each chunk. You can generate a dynamical power spectrum, for example, by taking an FFT of each chunk, or you can construct a convolution using a dot product. Some of these operations already exist in numpy and scipy, but others don't. One way to attack the problem would be to make a matrix in which each column was a starting location, and each row was a chunk. This would normally require duplicating some data, potentially a lot of data if there's a lot of overlap, but numpy's striding can be used to do this. The simplification of striding doesn't come for free; if you modify the array, all shared elements will be modified. Nevertheless, it's a useful operation. Find attached the code, `segmentaxis.py` . Example usage:

```
In [1]: import numpy as N
In [2]: import segmentaxis
In [3]: a = N.zeros(30)
In [4]: a[15] = 1
In [5]: filter = N.array([0.1,0.5,1,0.5,0.1])
In [6]: sa = segmentaxis.segment_axis(a,len(filter),len(filter)-1)
In [7]: sa
Out[7]:
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
In [8]: N.dot(sa[::2,:],filter)
Out[8]:
array([ 0\. ,  0\. ,  0\. ,  0\. ,  0\. ,  0\. ,  0.5,  0.5,  0\. ,
        0\. ,  0\. ])
In [9]: N.dot(sa[1::2,:],filter)
Out[9]:
array([ 0\. ,  0\. ,  0\. ,  0\. ,  0\. ,  0.1,  1\. ,  0.1,  0\. ,
        0\. ,  0\. ])
In [10]: N.dot(sa,filter)
Out[10]:
array([ 0\. ,  0\. ,  0\. ,  0\. ,  0\. ,  0\. ,  0\. ,  0\. ,  0\
        0.1,  0.5,  1\. ,  0.5,  0.1,  0\. ,  0\. ,  0\. ,  0\. ,
        0\. ,  0\. ,  0\. ,  0\. ])
```

Attachments

- `segmentaxis.py`

Indexing numpy arrays

The whole point of numpy is to introduce a multidimensional array object for holding homogeneously-typed numerical data. This is of course a useful tool for storing data, but it is also possible to manipulate large numbers of values without writing inefficient python loops. To accomplish this, one needs to be able to refer to elements of the arrays in many different ways, from simple “slices” to using arrays as lookup tables. The purpose of this page is to go over the various different types of indexing available. Hopefully the sometimes-peculiar syntax will also become more clear.

We will use the same arrays as examples wherever possible:

```
import numpy as np
A = np.arange(10)
```

```
A
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
B = np.reshape(np.arange(9),(3,3))
B
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
C = np.reshape(np.arange(2*3*4),(2,3,4))
C
```

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

Elements

The simplest way to pick one or some elements of an array looks very similar to python lists:

```
A[1]
```

```
1
```

```
B[1,0]
```

```
3
```

```
C[1,0,2]
```

```
14
```

That is, to pick out a particular element, you simply put the indices into square brackets after it. As is standard for python, element numbers start at zero.

If you want to change an array value in-place, you can simply use the syntax above in an assignment:

```
T = A.copy()
T[3] = -5
T
```

```
array([ 0,  1,  2, -5,  4,  5,  6,  7,  8,  9])
```

```
T[0] += 7
T
```

```
array([ 7,  1,  2, -5,  4,  5,  6,  7,  8,  9])
```

(The business with .copy() is to ensure that we don't actually modify A, since that would make further examples confusing.) Note that numpy also supports python's "augmented assignment" operators, +=, -=, *=, and so on.

Be aware that the type of array elements is a property of the array itself, so that if you try to assign an element of another type to an array, it will be silently converted (if possible):

```
T = A.copy()
T[3] = -1.5
T
```

```
array([ 0,  1,  2, -1,  4,  5,  6,  7,  8,  9])
```

```
T[3] = -0.5j
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent ca

<ipython-input-12-513dc3e86c66> in <module>()
----> 1 T[3] = -0.5j

TypeError: can't convert complex to long
```

```
T
```

```
array([ 0,  1,  2, -1,  4,  5,  6,  7,  8,  9])
```

Note that the conversion that happens is a default conversion; in the case of float to int conversion, it's truncation. If you wanted something different, say taking the floor, you would have to arrange that yourself (for example with np.floor()). In the case of converting complex values to integers, there's no resonable default way to do it, so numpy raises an exception and leaves the array unchanged.

Finally, two slightly more technical matters.

If you want to manipulate indices programmatically, you should know that when you write something like

```
C[1,0,1]
```

```
13
```

it is the same as (in fact it is internally converted to)

```
C[(1,0,1)]
```

```
13
```

This peculiar-looking syntax is constructing a tuple, python’s data structure for immutable sequences, and using that tuple as an index into the array. (Under the hood, C[1,0,1] is converted to C.**getitem**((1,0,1)).) This means you can whip up tuples if you want to:

```
i = (1,0,1)
C[i]
```

```
13
```

If it doesn’t seem likely you would ever want to do this, consider iterating over an arbitrarily multidimensional array:

```
for i in np.ndindex(B.shape):
    print i, B[i]
```

```
(0, 0) 0
(0, 1) 1
(0, 2) 2
(1, 0) 3
(1, 1) 4
(1, 2) 5
(2, 0) 6
(2, 1) 7
(2, 2) 8
```

Indexing with tuples will also become important when we start looking at fancy indexing and the function np.where().

The last technical issue I want to mention is that when you select an element from an array, what you get back has the same type as the array elements. This may sound obvious, and in a way it is, but keep in mind that even innocuous numpy

arrays like our A, B, and C often contain types that are not quite the python types:

```
a = C[1,2,3]
a
```

```
23
```

```
type(a)
```

```
numpy.int64
```

```
type(int(a))
```

```
int
```

```
a**a
```

```
-c:1: RuntimeWarning: overflow encountered in long_scalars
```

```
8450172506621111015
```

```
int(a)**int(a)
```

```
20880467999847912034355032910567L
```

numpy scalars also support certain indexing operations, for consistency, but these are somewhat subtle and under discussion.

Slices

It is obviously essential to be able to work with single elements of an array. But one of the selling points of numpy is the ability to do operations “array-wise”:

```
2*A
```

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

This is handy, but one very often wants to work with only part of an array. For example, suppose one wants to compute the array of differences of A, that is, the array whose elements are A[1]-A[0], A[2]-A[1], and so on. (In fact, the function np.diff does this, but let’s ignore that for expositional convenience.) numpy makes it possible to do this using array-wise operations:

```
A[1:]
```

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
A[:-1]
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

```
A[1:] - A[:-1]
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1])
```

This is done by making an array that is all but the first element of A, an array that is all but the last element of A, and subtracting the corresponding elements. The process of taking subarrays in this way is called “slicing”.

One-dimensional slices

The general syntax for a slice is *array*[*start*:*stop*:*step*]. Any or all of the values *start*, *stop*, and *step* may be left out (and if *step* is left out the colon in front of it may also be left out):

```
A[5:]
```

```
 array([5, 6, 7, 8, 9])
```

```
A[:5]
```

```
 array([0, 1, 2, 3, 4])
```

```
A[::2]
```

```
 array([0, 2, 4, 6, 8])
```

```
A[1::2]
```

```
 array([1, 3, 5, 7, 9])
```

```
A[1:8:2]
```

```
 array([1, 3, 5, 7])
```

As usual for python, the *start* index is included and the *stop* index is not included. Also as usual for python, negative numbers for *start* or *stop* count backwards from the end of the array:

```
A[-3:]
```

```
 array([7, 8, 9])
```

```
A[:-3]
```

```
 array([0, 1, 2, 3, 4, 5, 6])
```

If *stop* comes before *start* in the array, then an array of length zero is returned:

```
A[5:3]
```

```
array([], dtype=int64)
```

(The “dtype=int32” is present in the printed form because in an array with no elements, one cannot tell what type the elements have from their printed representation. It nevertheless makes sense to keep track of the type that they would have if the array had any elements.)

If you specify a slice that happens to have only one element, you get an array in return that happens to have only one element:

```
A[5:6]
```

```
array([5])
```

```
A[5]
```

```
5
```

This seems fairly obvious and reasonable, but when dealing with fancy indexing and multidimensional arrays it can be surprising.

If the number *step* is negative, the step through the array is negative, that is, the new array contains (some of) the elements of the original in reverse order:

```
A[::-1]
```

```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

This is extremely useful, but it can be confusing when *start* and *stop* are given:

```
A[5:3:-1]
```

```
array([5, 4])
```

```
A[3:5:1]
```

```
array([3, 4])
```

The rule to remember is: whether *step* is positive or negative, *start* is always included and *stop* never is.

Just as one can retrieve elements of an array as a subarray rather than one-by-one, one can modify them as a subarray rather than one-by-one:

```
#!python numbers=disable
>>> T = A.copy()
>>> T
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> T[1::2]
array([1, 3, 5, 7, 9])
>>> T[1::2] = -np.arange(5)
>>> T[1::2]
array([ 0, -1, -2, -3, -4])
>>> T
array([ 0,  0,  2, -1,  4, -2,  6, -3,  8, -4])
```

If the array you are trying to assign is the wrong shape, an exception is raised:

```
#!python numbers=disable
>>> T = A.copy()
>>> T[1::2] = np.arange(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single
>>> T[:4] = np.array([[0,1],[1,0]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single
```

If you think the error message sounds confusing, I have to agree, but there is a reason. In the first case, we tried to stuff six elements into five slots, so numpy refused. In the second case, there were the right number of elements - four - but we tried to stuff a two-by-two array where there was supposed to be a one-dimensional array of length four. While numpy could have coerced the two-by-two

array into the right shape, instead the designers chose to follow the python philosophy “explicit is better than implicit” and leave any coercing up to the user. Let’s do that, though:

```
#!python numbers=disable
>>> T = A.copy()
>>> T[:4] = np.array([[0,1],[1,0]]).ravel()
>>> T
array([0, 1, 1, 0, 4, 5, 6, 7, 8, 9])
```

So in order for assignment to work, it is not simply enough to have the right number of elements - they must be arranged in an array of the right shape.

There is another issue complicating the error message: numpy has some extremely convenient rules for converting lower-dimensional arrays into higher-dimensional arrays, and for implicitly repeating arrays along axes. This process is called “broadcasting”. We will see more of it elsewhere, but here it is in its simplest possible form:

```
#!python numbers=disable
>>> T = A.copy()
>>> T[1::2] = -1
>>> T
array([ 0, -1,  2, -1,  4, -1,  6, -1,  8, -1])
```

We told numpy to take a scalar, -1, and put it into an array of length five. Rather than signal an error, numpy’s broadcasting rules tell it to convert this scalar into an effective array of length five by repeating the scalar five times. (It does not, of course, actually create a temporary array of this size; in fact it uses a clever trick of telling itself that the temporary array has its elements spaced zero bytes apart.) This particular case of broadcasting gets used all the time:

```
#!python numbers=disable
>>> T = A.copy()
>>> T[1::2] -= 1
>>> T
array([0, 0, 2, 2, 4, 4, 6, 6, 8, 8])
```

Assignment is sometimes a good reason to use the “everything” slice:

```
#!python numbers=disable
>>> T = A.copy()
>>> T[:] = -1
>>> T
array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1])
>>> T = A.copy()
>>> T = -1
>>> T
-1
```

What happened here? Well, in the first case we told numpy to assign -1 to all the elements of T, so that's what it did. In the second case, we told python "T = -1". In python, variables are just names that can be attached to objects in memory. This is in sharp contrast with languages like C, where a variable is a named region of memory where data can be stored. Assignment to a variable name - T in this case - simply changes which object the name refers to, without altering the underlying object in any way. (If the name was the only reference to the original object, it becomes impossible for your program ever to find it again after the reassignment, so python deletes the original object to free up some memory.) In a language like C, assigning to a variable changes the value stored in that memory region. If you really must think in terms of C, you can think of all python variables as holding pointers to actual objects; assignment to a python variable is just modification of the pointer, and doesn't affect the object pointed to (unless garbage collection deletes it). In any case, if you want to modify the *contents* of an array, you can't do it by assigning to the name you gave the array; you must use slice assignment or some other approach.

Finally, a technical point: how can a program work with slices programmatically? What if you want to, say, save a slice specification to apply to many arrays later on? The answer is to use a slice object, which is constructed using slice():

```
#!python numbers=disable
>>> A[1::2]
array([1, 3, 5, 7, 9])
>>> s = slice(1,None,2)
>>> A[s]
array([1, 3, 5, 7, 9])
```

(Regrettably, you can't just write "s = 1::2". But within square brackets, 1::2 is converted internally to slice(1,None,2).) You can leave out arguments to slice() just like you can with the colon notation, with one exception:

```
#!python numbers=disable
>>> A[slice(-3)]
array([0, 1, 2, 3, 4, 5, 6])
>>> A[slice(None,3)]
array([0, 1, 2])
>>> A[slice()]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: slice expected at least 1 arguments, got 0
>>> A[slice(None,None,None)]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Multidimensional slices

One-dimensional arrays are extremely useful, but often one has data that is naturally multidimensional - image data might be an N by M array of pixel values, or an N by M by 3 array of colour values, for example. Just as it is useful to take slices of one-dimensional arrays, it is useful to take slices of multidimensional arrays. This is fairly straightforward:

```
#!python numbers=disable
>>> B
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> B[:2,:]
array([[0, 1, 2],
       [3, 4, 5]])
>>> B[:,::-1]
array([[2, 1, 0],
       [5, 4, 3],
       [8, 7, 6]])
```

Essentially one simply specifies a one-dimensional slice for each axis. One can also supply a number for an axis rather than a slice:

```
#!python numbers=disable
>>> B[0,:]
array([0, 1, 2])
>>> B[0,::-1]
array([2, 1, 0])
>>> B[:,0]
array([0, 3, 6])
```

Notice that when one supplies a number for (say) the first axis, the result is no longer a two-dimensional array; it's now a one-dimensional array. This makes sense:

```
#!python numbers=disable
>>> B[:,:]
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> B[0,:]
array([0, 1, 2])
>>> B[0,0]
0
```

If you supply no numbers, you get a two-dimensional array; if you supply one number, the dimension drops by one, and you get a one-dimensional array; and if you supply two numbers the dimension drops by two and you get a scalar. (If you think you should get a zero-dimensional array, you are opening a can of worms. The distinction, or lack thereof, between scalars and zero-dimensional arrays is an issue under discussion and development.)

If you are used to working with matrices, you may want to preserve a distinction between "row vectors" and "column vectors". numpy supports only one kind of one-dimensional array, but you could represent row and column vectors as *two*-dimensional arrays, one of whose dimensions happens to be one. Unfortunately indexing of these objects then becomes cumbersome.

As with one-dimensional arrays, if you specify a slice that happens to have only one element, you get an array one of whose axes has length 1 - the axis doesn't "disappear" the way it would if you had provided an actual number for that axis:

```
#!python numbers=disable
>>> B[:,0:1]
array([[0],
       [3],
       [6]])
>>> B[:,0]
array([0, 3, 6])
```

numpy also has a few shortcuts well-suited to dealing with arrays with an indeterminate number of dimensions. If this seems like something unreasonable, keep in mind that many of numpy's functions (for example np.sort(), np.sum(), and np.transpose()) must work on arrays of arbitrary dimension. It is of course possible to extract the number of dimensions from an array and work with it explicitly, but one's code tends to fill up with things like (slice(None,None,None),)*(C.ndim-1), making it unpleasant to read. So numpy has some shortcuts which often simplify things.

First the Ellipsis object:

```
#!python numbers=disable
>>> A[...]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> B[...]
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> B[0,...]
array([0, 1, 2])
>>> B[0,...,0]
array(0)
>>> C[0,...,0]
array([0, 4, 8])
>>> C[0,Ellipsis,0]
array([0, 4, 8])
```

The ellipsis (three dots) indicates “as many ‘:’ as needed”. (Its name for use in index-fiddling code is Ellipsis, and it’s not numpy-specific.) This makes it easy to manipulate only one dimension of an array, letting numpy do array-wise operations over the “unwanted” dimensions. You can only really have one ellipsis in any given indexing expression, or else the expression would be ambiguous about how many ‘:’ should be put in each. (In fact, for some reason it is allowed to have something like “C[...,...]”; this is not actually ambiguous.)

In some circumstances, it is convenient to omit the ellipsis entirely:

```
#!python numbers=disable
>>> B[0]
array([0, 1, 2])
>>> C[0]
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> C[0,0]
array([0, 1, 2, 3])
>>> B[0:2]
array([[0, 1, 2],
       [3, 4, 5]])
```

If you don’t supply enough indices to an array, an ellipsis is silently appended. This means that in some sense you can view a two-dimensional array as an array of one-dimensional arrays. In combination with numpy’s array-wise operations, this means that functions written for one-dimensional arrays can often just work for two-dimensional arrays. For example, recall the difference operation we wrote out in the section on one-dimensional slices:

```
#!python numbers=disable
>>> A[1:] - A[:-1]
array([1, 1, 1, 1, 1, 1, 1, 1, 1])
>>> B[1:] - B[:-1]
array([[3, 3, 3],
       [3, 3, 3]])
```

It works, unmodified, to take the differences along the first axis of a two-dimensional array.

Writing to multidimensional slices works just the way writing to one-dimensional slices does:

```
>>> T = B.copy()
>>> T[1,:] = -1
>>> T
array([[ 0,  1,  2],
 [-1, -1, -1],
 [ 6,  7,  8]])
>>> T[:,:2] = -2
>>> T
array([[-2, -2,  2],
 [-2, -2, -1],
 [-2, -2,  8]])
```

FIXME: np.newaxis and broadcasting rules.

Views versus copies

FIXME: Zero-dimensional arrays, views of a single element.

Fancy indexing

Slices are very handy, and the fact that they can be created as views makes them efficient. But some operations cannot really be done with slices; for example, suppose one wanted to square all the negative values in an array. Short of writing a loop in python, one wants to be able to locate the negative values, extract them, square them, and put the new values where the old ones were:

```
#!python numbers=disable
>>> T = A.copy() - 5
>>> T[T<0] **= 2
>>> T
array([25, 16,  9,  4,  1,  0,  1,  2,  3,  4])
```

Or suppose one wants to use an array as a lookup table, that is, for an array B, produce an array whose i,j th element is LUT[B[i,j]]: FIXME: argsort is a better example

```
#!python numbers=disable
>>> LUT = np.sin(A)
>>> LUT
array([ 0\.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.4121
>>> LUT[B]
array([[ 0\.        ,  0.84147098,  0.90929743],
       [ 0.14112001, -0.7568025 , -0.95892427],
       [-0.2794155 ,  0.6569866 ,  0.98935825]])
```

For this sort of thing numpy provides what is called “fancy indexing”. It is not nearly as quick and lightweight as slicing, but it allows one to do some rather sophisticated things while letting numpy do all the hard work in C.

Boolean indexing

It frequently happens that one wants to select or modify only the elements of an array satisfying some condition. numpy provides several tools for working with this sort of situation. The first is boolean arrays. Comparisons - equal to, less than, and so on - between numpy arrays produce arrays of boolean values:

```
#!python numbers=disable
>>> A<5
array([ True,  True,  True,  True,  True, False, False, False, Fals
```

These are normal arrays. The actual storage type is normally a single byte per value, not bits packed into a byte, but boolean arrays offer the same range of indexing and array-wise operations as other arrays. Unfortunately, python’s “and” and “or” cannot be overridden to do array-wise operations, so you must use the bitwise operations “&”, “|”, and “^” (for exclusive-or). Similarly python’s chained inequalities cannot be overridden. Also, regrettably, one cannot chage the precence of the bitwise operators:

```
#!python numbers=disable
>>> c = A<5 & A>1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: The truth value of an array with more than one element
>>> c = (A<5) & (A>1)
>>> c
array([False, False,  True,  True,  True, False, False, False, Fals
```

Nevertheless, numpy’s boolean arrays are extremely powerful.

One can use boolean arrays to extract values from arrays:

```
#!python numbers=disable
>>> c = (A<5) & (A>1)
>>> A[c]
array([2, 3, 4])
```

The result is necessarily a copy of the original array, rather than a view, since it will not normally be the case the the elements of c that are True select an evenly-strided memory layout. Nevertheless it is also possible to use boolean arrays to write to specific elements:

```
>>> T = A.copy()
>>> c = (A<5) & (A>1)
>>> T[c] = -7
>>> T
array([ 0,  1, -7, -7, -7,  5,  6,  7,  8,  9])
```

FIXME: mention where()

Multidimensional boolean indexing

Boolean indexing works for multidimensional arrays as well. In its simplest (and most common) incarnation, you simply supply a single boolean array as index, the same shape as the original array:

```
>>> C[C%5==0]
array([ 0,  5, 10, 15, 20])
```

You then get back a one-dimensional array of the elements for which the condition is True. (Note that the array must be one-dimensional, since the boolean values can be arranged arbitrarily around the array. If you want to keep track of the

arrangement of values in the original array, look into using numpy's "masked array" tools.) You can also use boolean indexing for assignment, just as you can for one-dimensional arrays.

Two very useful operations on boolean arrays are np.any() and np.all():

```
>>> np.any(B<5)
True
>>> np.all(B<5)
False
```

They do just what they say on the tin, evaluate whether any entry in the boolean matrix is True, or whether all elements in the boolean matrix are True. But they can also be used to evaluate "along an axis", for example, to produce a boolean array saying whether any element in a given row is True:

```
>>> B<5
array([[ True,  True,  True],
 [ True,  True, False],
 [False, False, False]], dtype=bool)
>>> np.any(B<5, axis=1)
array([ True,  True, False], dtype=bool)
>>> np.all(B<5, axis=1)
array([ True, False, False], dtype=bool)
```

One can also use boolean indexing to pull out rows or columns meeting some criterion:

```
>>> B[np.any(B<5, axis=1),:]
array([[0, 1, 2],
 [3, 4, 5]])
```

The result here is two-dimensional because there is one dimension for the results of the boolean indexing, and one dimension because each row is one-dimensional.

This works with higher-dimensional boolean arrays as well:

```
>>> c = np.any(C<5,axis=2)
>>> c
array([[ True,  True, False],
 [False, False, False]], dtype=bool)
>>> C[c,:]
array([[0, 1, 2, 3],
 [4, 5, 6, 7]])
```

Here too the result is two-dimensional, though that is perhaps a little more surprising. The boolean array is two-dimensional, but the part of the return value corresponding to the boolean array must be one-dimensional, since the True values may be distributed arbitrarily. The subarray of C corresponding to each True or False value is one-dimensional, so we get a return array of dimension two.

Finally, if you want to apply boolean conditions to the rows and columns simultaneously, beware:

```
>>> B[np.array([True, False, True]), np.array([False, True, True])]
array([1, 8])
>>> B[np.array([True, False, True]),:][:,np.array([False, True, Tru
array([[1, 2],
 [7, 8]])
```

The obvious approach doesn't give the right answer. I don't know why not, or why it produces the value that it does. You can get the right answer by indexing twice, but that's clumsy and inefficient and doesn't allow assignment.

FIXME: works with too-small boolean arrays for some reason?

List-of-locations indexing

It happens with some frequency that one wants to pull out values at a particular location in an array. If one wants a single location, one can just use simple indexing. But if there are many locations, you need something a bit more clever. Fortunately numpy supports a mode of fancy indexing that accomplishes this:

```
>>> primes = np.array([2,3,5,7,11,13,17,19,23])
>>> idx = [3,4,1,2,2]
>>> primes[idx]
array([ 7, 11,  3,  5,  5])
>>> idx = np.array([3,4,1,2,2])
>>> primes[idx]
array([ 7, 11,  3,  5,  5])
```

When you index with an array that is not an array of booleans, or with a list, numpy views it as an array of indices. The array can be any shape, and the returned array has the same shape:

```
>>> primes = np.array([2,3,5,7,11,13,17,19,23,29,31])
>>> primes[B]
array([[ 2,  3,  5],
 [ 7, 11, 13],
 [17, 19, 23]])
```

Effectively this uses the original array as a look-up table.

You can also assign to arrays in this way:

```
>>> T = A.copy()
>>> T[ [1,3,5,0] ] = -np.arange(4)
>>> T
array([-3,  0,  2, -1,  4, -2,  6,  7,  8,  9])
```

Warning: Augmented assignment - the operators like “+=” - works, but it does not necessarily do what you would expect. In particular, repeated indices do not result in the value getting added twice:

```
>>> T = A.copy()
>>> T[ [0,1,2,3,3,3] ] += 10
>>> T
array([10, 11, 12, 13,  4,  5,  6,  7,  8,  9])
```

This is surprising, inconvenient, and unfortunate, but it is a direct result of how python implements the “+=” operators. The most common case for doing this is something histogram-like:

```
>>> bins = np.zeros(5,dtype=np.int32)
>>> pos = [1,0,2,0,3]
>>> wts = [1,2,1,1,4]
>>> bins[pos]+=wts
>>> bins
array([1, 1, 1, 4, 0])
```

Unfortunately this gives the wrong answer. In older versions of numpy there was no really satisfactory solution, but as of numpy 1.1, the histogram function can do this:

```
>>> bins = np.zeros(5,dtype=np.int32)
>>> pos = [1,0,2,0,3]
>>> wts = [1,2,1,1,4]
>>> np.histogram(pos,bins=5,range=(0,5),weights=wts,new=True)
(array([3, 1, 1, 4, 0]), array([ 0.,  1.,  2.,  3.,  4.,  5.]))
```

FIXME: mention put() and take()

Multidimensional list-of-locations indexing

One can also, not too surprisingly, use list-of-locations indexing on multidimensional arrays. The syntax is, however, a bit surprising. Let's suppose we want the list [B[0,0],B[1,2],B[0,1]]. Then we write:

```
>>> B[ [0,1,0], [0,2,1] ]
array([0, 5, 1])
>>> [B[0,0],B[1,2],B[0,1]]
[0, 5, 1]
```

This may seem weird - why not provide a list of tuples representing coordinates? Well, the reason is basically that for large arrays, lists and tuples are very inefficient, so numpy is designed to work with arrays only, for indices as well as values. This means that something like B[[(0,0),(1,2),(0,1)]] looks just like indexing B with a two-dimensional array, which as we saw above just means that B should be used as a look-up table yielding a two-dimensional array of results (each of which is one-dimensional, as usual when we supply only one index to a two-dimensional array).

In summary, in list-of-locations indexing, you supply an array of values for each coordinate, all the same shape, and numpy returns an array of the same shape containing the values obtained by looking up each set of coordinates in the original array. If the coordinate arrays are not the same shape, numpy's broadcasting rules are applied to them to try to make their shapes the same. If there are not as many arrays as the original array has dimensions, the original array is regarded as containing arrays, and the extra dimensions appear on the result array.

Fortunately, most of the time when one wants to supply a list of locations to a multidimensional array, one got the list from numpy in the first place. A normal way to do this is something like:

```
>>> idx = np.nonzero(B%2)
>>> idx
(array([0, 1, 1, 2]), array([1, 0, 2, 1]))
>>> B[idx]
array([1, 3, 5, 7])
>>> B[B%2 != 0]
array([1, 3, 5, 7])
```

Here nonzero() takes an array and returns a list of locations (in the correct format) where the array is nonzero. Of course, one can also index directly into the array with a boolean array; this will be much more efficient unless the number of nonzero locations is small and the indexing is done many times. But sometimes it is valuable to work with the list of indices directly.

Picking out rows and columns

One unfortunate consequence of numpy's list-of-locations indexing syntax is that users used to other array languages expect it to pick out rows and columns. After all, it's quite reasonable to want to pull out a list of rows and columns from a matrix. So numpy provides a convenience function, ix_() for doing this:

```
>>> B[ np.ix_([0,2],[0,2]) ]
array([[0, 2],
 [6, 8]])
>>> np.ix_([0,2],[0,2])
(array([[0],
 [2]]), array([[0, 2]]))
```

The way it works is by taking advantage of numpy's broadcasting facilities. You can see that the two arrays used as row and column indices have different shapes; numpy's broadcasting repeats each along the too-short axis so that they conform.

Mixed indexing modes

What happens when you try to mix slice indexing, element indexing, boolean indexing, and list-of-locations indexing?

How indexing works under the hood

A numpy array is a block of memory, a data type for interpreting memory locations, a list of sizes, and a list of strides. So for example, C[i,j,k] is the element starting at position i*strides[0]+j*strides[1]+k*strides[2]. This means, for example, that transposing amatrix can be done very efficiently: just reverse the strides and sizes arrays. This is why slices are efficient and can return views, but fancy indexing is slower and can't.

At a python level, numpy's indexing works by overriding the **getitem** and **setitem** methods in an ndarray object. These methods are called when arrays are indexed, and they allow arbitrary implementations:

```
>>> class IndexDemo:
...     def __getitem__(self, *args):
...         print "__getitem__", args
...         return 1
...     def __setitem__(self, *args):
...         print "__setitem__", args
...     def __iadd__(self, *args):
...         print "__iadd__", args
...
>>>
>>> T = IndexDemo()
>>> T[1]
__getitem__ (1,)
1
>>> T["fish"]
__getitem__ ('fish',)
1
>>> T[A]
__getitem__ (array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),)
1
>>> T[1,2]
__getitem__ ((1, 2),)
1
>>> T[1] = 7
__setitem__ (1, 7)
>>> T[1] += 7
__getitem__ (1,)
__setitem__ (1, 8)
```

Array-like objects

numpy and scipy provide a few other types that behave like arrays, in particular matrices and sparse matrices. Their indexing can differ from that of arrays in surprising ways.

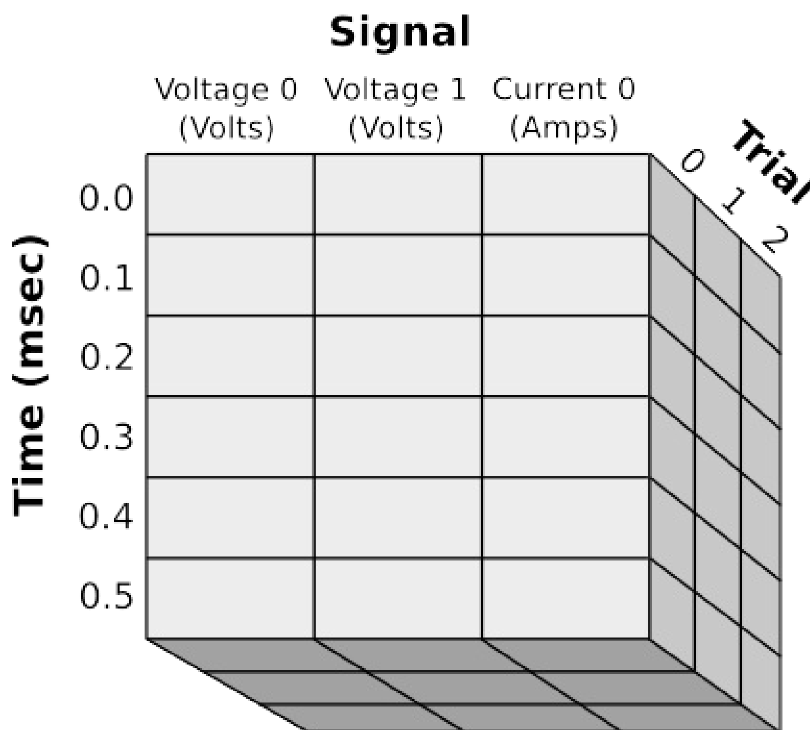
MetaArray

!MetaArray is a class that extends ndarray, adding support for per-axis meta data storage. This class is useful for storing data arrays along with units, axis names, column names, axis values, etc. !MetaArray objects can be indexed and sliced arbitrarily using named axes and columns.

Download here: `MetaArray.py`

Example Uses

Here is an example of the type of data one might store with !MetaArray:



Notice that each axis is named and can store different types of meta information: *The Signal axis has named columns with different units for each column* The Time axis associates a numerical value with each row * The Trial axis uses normal integer indexes

Data from this array can be accessed many different ways:

```
data[0, 1, 1]
data[:, "Voltage 1", 0]
data["Trial":1, "Signal":"Voltage 0"]
data["Time":slice(3,7)]
```

Features

```
* Per axis meta-information:`` * Named axes`` * Numerical values wi
```

Documentation

Instantiation

```
Accepted Syntaxes:
```

```
 # Constructs MetaArray from a preexisting ndarray and info list
MetaArray(ndarray, info)

 # Constructs MetaArray using empty(shape, dtype=type) and info lis
MetaArray((shape), dtype=type, info)

 # Constructs MetaArray from file written using MetaArray.write()
MetaArray(file='fileName')
```

```
info parameter: This parameter specifies the entire set of meta data
```

```
info=[axis1, axis2, axis3...]
```

```
Each axis description is a dict which may contain:`` * "name": the n
```

```
For example, the data set shown above would look like:
```

```
MetaArray((3, 6, 3), dtype=float, info=[
  {"name": "Signal", "cols": [
      {"name": "Voltage 0", "units": "V"},
      {"name": "Voltage 1", "units": "V"},
      {"name": "Current 0", "units": "A"}
    ]
  },
  {"name": "Time", "units": "msec", "values":[0.0, 0.1, 0.2, 0.3, 0
  {"name": "Trial"},
  {"note": "Just some extra info"}
]
```

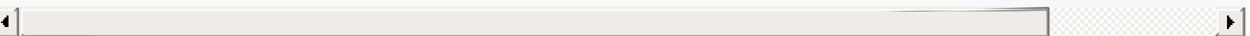
Accessing Data

Data can be accessed through a variety of methods: *Standard indexing–You may always just index the array exactly as you would any ndarray* Named axes–If you don’t remember the order of axes, you may specify the axis to be indexed or sliced like this:

```
data["AxisName":index]
data["AxisName":slice(...)]
```

```
Note that since this syntax hijacks the original slice mechanism, yo
```

```
data["AxisName":"ColumnName"]
data["ColumnName"]  ## Works only if the named column exists for th
data[["ColumnName1", "ColumnName2"]]
```



```
* Boolean selection--works as you might normally expect, for example
```

```
sel = data["ColumnName", 0, 0] > 0.2
data[sel]
```

```
* Access axis values using !MetaArray.axisValues(), or .xvals() for
```

File I/O

```
data.write('fileName')
newData = MetaArray(file='fileName')
```

Performance Tips

!MetaArray is a subclass of ndarray which overrides the `__getitem__` and `__setitem__` methods. Since these methods must alter the structure of the meta information for each access, they are quite slow compared to the native methods. As a result, many builtin functions will run very slowly when operating on a !MetaArray. It is recommended, therefore, that you recast your arrays before performing these operations like this:

```
data = MetaArray(...)
data.mean()                ## Very slow
data.view(ndarray).mean()  ## native speed
```

More Examples

```
A 2D array of altitude values for a topographical map might look lil
```

```
info=[
  {'name': 'lat', 'title': 'Latitude'},
  {'name': 'lon', 'title': 'Longitude'},
  {'title': 'Altitude', 'units': 'm'}
]
```

```
In this case, every value in the array represents the altitude in fe
```

```
array[10, 5]
array['lon':5, 'lat':10]
array['lat':10][5]
```

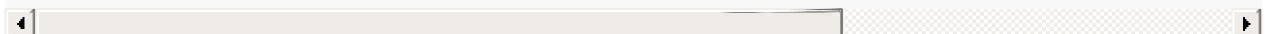
```
Now suppose we want to combine this data with another array of equal
```

```
info=[
  {'name': 'vals', 'cols': [
    {'name': 'altitude', 'units': 'm'},
    {'name': 'rainfall', 'units': 'cm/year'}
  ]},
  {'name': 'lat', 'title': 'Latitude'},
  {'name': 'lon', 'title': 'Longitude'}
]
```

```
We can now access the altitude values with array[0] or array['altitu
```

```
array[1, 10, 5]
array['lon':5, 'lat':10, 'val': 'rainfall']
array['rainfall', 'lon':5, 'lat':10]
```

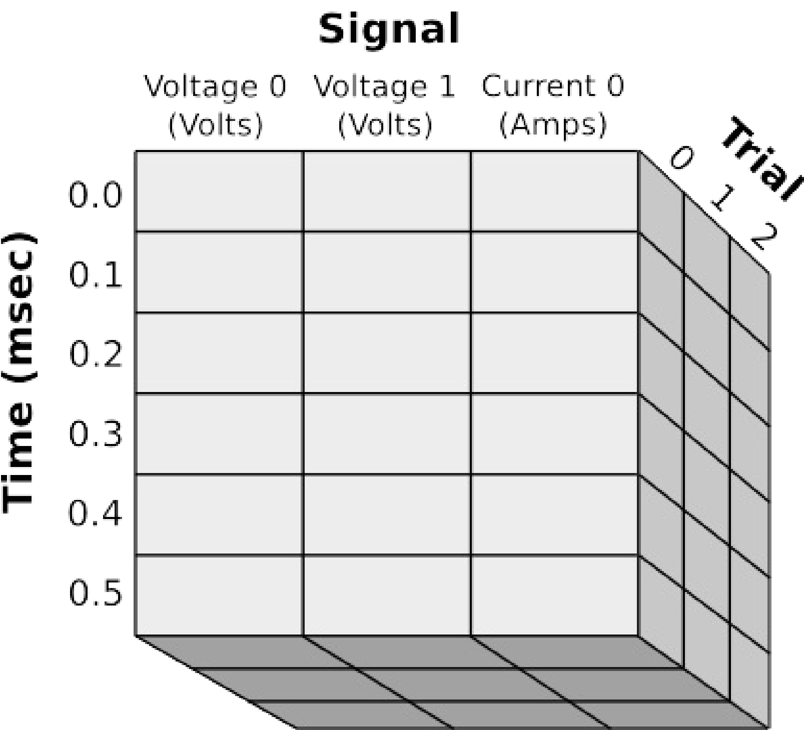
```
Notice that in the second example, there is no need for an extra (4
since the actual values are described (name and units) in the colum
```



=== Contact === Luke Campagnola - lcampagn@email.unc.edu

Attachments

- `MetaArray.py`
- `example.png`


Signal
Voltage 0 (Volts)
Voltage 1 (Volts)
Current 0 (Amps)
Trial
0
1
2
Time (msec)
0.0
0.1
0.2
0.3
0.4
0.5

Multidot

The matrix multiplication function, numpy.dot(), only takes two arguments. That means to multiply more than two arrays together you end up with nested function calls which are hard to read:

```
dot(dot(dot(a,b),c),d)
```

versus infix notation where you'd just be able to write

```
a*b*c*d
```

There are a couple of ways to define an 'mdot' function that acts like dot but accepts more than two arguments. Using one of these allows you to write the above expression as

```
mdot(a,b,c,d)
```

Using reduce

The simplest way it to just use reduce.

```
def mdot(*args):
    return reduce(numpy.dot, args)
```

Or use the equivalent loop (which is apparently the preferred style for Py3K):

```
def mdot(*args):
    ret = args[0]
    for a in args[1:]:
        ret = dot(ret,a)
    return ret
```

This will always give you left to right associativity, i.e. the expression is interpreted as `(((a*b)*c)*d)` .

You also can make a right-associative version of the loop:

```
def mdotr(*args):
    ret = args[-1]
    for a in reversed(args[:-1]):
        ret = dot(a,ret)
    return ret
```

which evaluates as `(a*(b*(c*d)))`. But sometimes you'd like to have finer control since the order in which matrix multiplies are performed can have a big impact on performance. The next version gives that control.

Controlling order of evaluation

If we're willing to sacrifice Numpy's ability to treat tuples as arrays, we can use tuples as grouping constructs. This version of `mdot` allows syntax like this:

```
mdot(a,((b,c),d))
```

to control the order in which the pairwise `dot` calls are made.

```
import types
import numpy
def mdot(*args):
   """Multiply all the arguments using matrix product rules.
  The output is equivalent to multiplying the arguments one by one
  from left to right using dot().
  Precedence can be controlled by creating tuples of arguments,
  for instance mdot(a,((b,c),d)) multiplies a (a*((b*c)*d)).
  Note that this means the output of dot(a,b) and mdot(a,b) will dif
  a or b is a pure tuple of numbers.
  """
   if len(args)==1:
       return args[0]
   elif len(args)==2:
       return _mdot_r(args[0],args[1])
   else:
       return _mdot_r(args[:-1],args[-1])

def _mdot_r(a,b):
   """Recursive helper for mdot"""
   if type(a)==types.TupleType:
       if len(a)>1:
           a = mdot(*a)
       else:
           a = a[0]
   if type(b)==types.TupleType:
       if len(b)>1:
           b = mdot(*b)
       else:
           b = b[0]
   return numpy.dot(a,b)
```

Multiply

Note that the elementwise multiplication function `numpy.multiply` has the same two-argument limitation as `numpy.dot`. The exact same generalized forms can be defined for multiply.

Left associative versions:

```
def mmultiply(*args):
    return reduce(numpy.multiply, args)
```

```
def mmultiply(*args):
    ret = args[0]
    for a in args[1:]:
        ret = multiply(ret,a)
    return ret
```

Right-associative version:

```
def mmultiplyr(*args):
    ret = args[-1]
    for a in reversed(args[:-1]):
        ret = multiply(a,ret)
    return ret
```

Version using tuples to control order of evaluation:

```
import types
import numpy
def mmultiply(*args):
   """Multiply all the arguments using elementwise product.
 The output is equivalent to multiplying the arguments one by one
 from left to right using multiply().
 Precedence can be controlled by creating tuples of arguments,
 for instance mmultiply(a,((b,c),d)) multiplies a (a*((b*c)*d)).
 Note that this means the output of multiply(a,b) and mmultiply(a,b
 a or b is a pure tuple of numbers.
 """
   if len(args)==1:
       return args[0]
   elif len(args)==2:
       return _mmultiply_r(args[0],args[1])
   else:
       return _mmultiply_r(args[:-1],args[-1])

def _mmultiply_r(a,b):
   """Recursive helper for mmultiply"""
   if type(a)==types.TupleType:
       if len(a)>1:
           a = mmultiply(*a)
       else:
           a = a[0]
   if type(b)==types.TupleType:
       if len(b)>1:
           b = mmultiply(*b)
       else:
           b = b[0]
   return numpy.multiply(a,b)
```

Object arrays using record arrays

numpy supports working with arrays of python objects, but these arrays lack the type-uniformity of normal numpy arrays, so they can be quite inefficient in terms of space and time, and they can be quite cumbersome to work with. However, it would often be useful to be able to store a user-defined class in an array.

One approach is to take advantage of numpy's record arrays. These are arrays in which each element can be large, as it has named and typed fields; essentially they are numpy's equivalent to arrays of C structures. Thus if one had a class consisting of some data - named fields, each of a numpy type - and some methods, one could represent the data for an array of these objects as a record array. Getting the methods is more tricky.

One approach is to create a custom subclass of the numpy array which handles conversion to and from your object type. The idea is to store the data for each instance internally in a record array, but when indexing returns a scalar, construct a new instance from the data in the records. Similarly, when assigning to a particular element, the array subclass would convert an instance to its representation as a record.

Attached is an implementation of the above scheme.

Attachments

- `obarray.py`
- `test_obarray.py`

Stride tricks for the Game of Life

This is similar to [:../SegmentAxis:Segment axis], but for 2D arrays with 2D windows.

The Game of Life is a cellular automaton devised by the British mathematician John Horton Conway in 1970, see [1].

It consists of a rectangular grid of cells which are either dead or alive, and a transition rule for updating the cells' state. To update each cell in the grid, the state of the 8 neighbouring cells needs to be examined, i.e. it would be desirable to have an easy way of accessing the 8 neighbours of all the cells at once without making unnecessary copies. The code snippet below shows how to use the devious stride tricks for that purpose.

[1] Game of Life) at Wikipedia

```
import numpy as np
from numpy.lib import stride_tricks
x = np.arange(20).reshape([4, 5])
xx = stride_tricks.as_strided(x, shape=(2, 3, 3, 3), strides=x.str
```

```
x
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

```
xx
```

```
array([[[[ 0,  1,  2],
         [ 5,  6,  7],
         [10, 11, 12]],

        [[ 1,  2,  3],
         [ 6,  7,  8],
         [11, 12, 13]],

        [[ 2,  3,  4],
         [ 7,  8,  9],
         [12, 13, 14]]],

       [[[ 5,  6,  7],
         [10, 11, 12],
         [15, 16, 17]],

        [[ 6,  7,  8],
         [11, 12, 13],
         [16, 17, 18]],

        [[ 7,  8,  9],
         [12, 13, 14],
         [17, 18, 19]]]])
```

```
xx[0,0]
```

```
array([[ 0,  1,  2],
       [ 5,  6,  7],
       [10, 11, 12]])
```

```
xx[1,2]
```

```
array([[ 7,  8,  9],
       [12, 13, 14],
       [17, 18, 19]])
```

```
x.strides
```

```
(40, 8)
```

```
xx.strides
```

```
(40, 8, 40, 8)
```

accumarray like function

accum, a function like MATLAB’s accumarray

NumPy doesn’t include a function that is equivalent to MATLAB’s `accumarray` function. The following function, `accum`, is close.

Note that `accum` can handle n-dimensional arrays, and allows the data type of the result to be specified.

```
from itertools import product
import numpy as np

def accum(accmap, a, func=None, size=None, fill_value=0, dtype=None
    """
 An accumulation function similar to Matlab's `accumarray` functio

 Parameters
 ----------
 accmap : ndarray
 This is the "accumulation map".  It maps input (i.e. indices into
 `a`) to their destination in the output array.  The first `a.ndim
 dimensions of `accmap` must be the same as `a.shape`.  That is,
 `accmap.shape[:a.ndim]` must equal `a.shape`.  For example, if `a
 has shape (15,4), then `accmap.shape[:2]` must equal (15,4).  In t
 case `accmap[i,j]` gives the index into the output array where
 element (i,j) of `a` is to be accumulated.  If the output is, say,
 a 2D, then `accmap` must have shape (15,4,2).  The value in the
 last dimension give indices into the output array. If the output i
 1D, then the shape of `accmap` can be either (15,4) or (15,4,1)
 a : ndarray
 The input data to be accumulated.
 func : callable or None
 The accumulation function.  The function will be passed a list
 of values from `a` to be accumulated.
 If None, numpy.sum is assumed.
 size : ndarray or None
 The size of the output array.  If None, the size will be determine
 from `accmap`.
 fill_value : scalar
 The default value for elements of the output array.
 dtype : numpy data type, or None
 The data type of the output array.  If None, the data type of
 `a` is used.

 Returns
 -------
 out : ndarray
```

```
The accumulated results.

The shape of `out` is `size` if `size` is given.  Otherwise the
shape is determined by the (lexicographically) largest indices of
the output found in `accmap`.

Examples
--------
>>> from numpy import array, prod
>>> a = array([[1,2,3],[4,-1,6],[-1,8,9]])
>>> a
array([[ 1,  2,  3],
[ 4, -1,  6],
[-1,  8,  9]])
>>> # Sum the diagonals.
>>> accmap = array([[0,1,2],[2,0,1],[1,2,0]])
>>> s = accum(accmap, a)
array([9, 7, 15])
>>> # A 2D output, from sub-arrays with shapes and positions like
>>> # [ (2,2) (2,1)]
>>> # [ (1,2) (1,1)]
>>> accmap = array([
[[0,0],[0,0],[0,1]],
[[0,0],[0,0],[0,1]],
[[1,0],[1,0],[1,1]],
])
>>> # Accumulate using a product.
>>> accum(accmap, a, func=prod, dtype=float)
array([[ -8.,  18.],
[ -8.,   9.]])
>>> # Same accmap, but create an array of lists of values.
>>> accum(accmap, a, func=lambda x: x, dtype='O')
array([[[1, 2, 4, -1], [3, 6]],
[[-1, 8], [9]]], dtype=object)
"""

   # Check for bad arguments and handle the defaults.
   if accmap.shape[:a.ndim] != a.shape:
       raise ValueError("The initial dimensions of accmap must be
   if func is None:
       func = np.sum
   if dtype is None:
       dtype = a.dtype
   if accmap.shape == a.shape:
       accmap = np.expand_dims(accmap, -1)
   adims = tuple(range(a.ndim))
   if size is None:
       size = 1 + np.squeeze(np.apply_over_axes(np.max, accmap, ax
   size = np.atleast_1d(size)

   # Create an array of python lists of values.
   vals = np.empty(size, dtype='O')
   for s in product(*[range(k) for k in size]):
```

```
        vals[s] = []
    for s in product(*[range(k) for k in a.shape]):
        indx = tuple(accmap[s])
        val = a[s]
        vals[indx].append(val)

    # Create the output array.
    out = np.empty(size, dtype=dtype)
    for s in product(*[range(k) for k in size]):
        if vals[s] == []:
            out[s] = fill_value
        else:
            out[s] = func(vals[s])

    return out
```

Examples

A basic example–sum the diagonals (with wrapping) of a 3 by 3 array:

```
from numpy import array, prod

a = array([[1,2,3],[4,-1,6],[-1,8,9]])
accmap = array([[0,1,2],[2,0,1],[1,2,0]])
```

```
s = accum(accmap, a)
s
```

```
 array([ 9,  7, 15])
```

Accumulate using multiplication, going from a 3 by 3 array to 2 by 2 array:

```
accmap = array([
             [[0,0],[0,0],[0,1]],
             [[0,0],[0,0],[0,1]],
             [[1,0],[1,0],[1,1]],
         ])

accum(accmap, a, func=prod, dtype=float)
```

```
array([[ -8.,  18.],
       [ -8.,   9.]])
```

Create an array of lists containing the values to be accumulated in each position in the output array:

```
accum(accmap, a, func=lambda x: x, dtype='O')
```

```
array([[[1, 2, 4, -1], [3, 6]],
       [[-1, 8], [9]]], dtype=object)
```

Use `accum` to arrange some values from a 1D array in a 2D array (note that using `accum` for this is overkill; fancy indexing would suffice):

```
subs = np.array([[k,5-k] for k in range(6)])
subs
```

```
array([[0, 5],
       [1, 4],
       [2, 3],
       [3, 2],
       [4, 1],
       [5, 0]])
```

```
vals = array(range(10,16))
accum(subs, vals)
```

```
array([[ 0,  0,  0,  0,  0, 10],
       [ 0,  0,  0,  0, 11,  0],
       [ 0,  0,  0, 12,  0,  0],
       [ 0,  0, 13,  0,  0,  0],
       [ 0, 14,  0,  0,  0,  0],
       [15,  0,  0,  0,  0,  0]])
```

Other examples

- C Extensions for Using NumPy Arrays
- Embedding in Traits GUI
- Matplotlib: drag'n'drop text example
- Matplotlib: treemap
- Mayavi: Install python stuff from source
- Mayavi: examples
- Reading custom text files with Pyparsing
- Scripting Mayavi 2: basic modules
- Scripting Mayavi 2: filters
- Scripting Mayavi 2: main modules

C Extensions for Using NumPy Arrays

I've written several C extensions that handle NumPy arrays. They are simple, but they seem to work well. They will show you how to pass Python variables and NumPy arrays to your C code. Once you learn how to do it, it's pretty straight-forward. I suspect they will suffice for most numerical code. I've written it up as a draft and have made the code and document file available. I found for my numerical needs I really only need to pass a limited set of things (integers, floats, strings, and NumPy arrays). If that's your category, this code might help you.

I have tested the routines and so far,so good, but I cannot guarantee anything. I am a bit new to this. If you find any errors put up a message on the SciPy mailing list.

A link to the tar ball that holds the code and docs is given below.

I have recently updated some information and included more examples. The document presented below is the original documentation which is still useful. The link below holds the latest documentation and source code.

- `Cext_v2.tar.gz`

– Lou Pecora

C Extensions to NumPy and Python

By Lou Pecora - 2006-12-07 (Draft version 0.1)

Overview

Introduction– a little background

In my use of Python I came across a typical problem: I needed to speed up particular parts of my code. I am not a Python guru or any kind of coding/computer guru. I use Python for numerical calculations and I make heavy use of Numeric/NumPy. Almost every Python book or tutorial tells you build C extensions to Python when you need a routine to run fast. C extensions are C code that can be compiled and linked to a shared library that can be imported like any Python module and you can call specified C routines like they were Python functions.

Sounds nice, but I had reservations. It looked non-trivial (it is, to an extent). So I searched for other solutions. I found them. They are such approaches as SWIG, Pyrex, ctypes, Psyco, and Weave. I often got the simple examples given to work (not all, however) when I tried these. But I hit a barrier when I tried to apply them to NumPy. Then one gets into typemaps or other hybrid constructs. I am not

knocking these approaches, but I could never figure them out and get going on my own code despite lots of online tutorials and helpful suggestions from various Python support groups and emailing lists.

So I decided to see if I could just write my own C extensions. I got help in the form of some simple C extension examples for using Numeric written about 2000 from Tom Loredo of Cornell university. These sat on my hard drive until 5 years later out of desperation I pulled them out and using his examples, I was able to quickly put together several C extensions that (at least for me) handle all of the cases (so far) where I want a speedup. These cases mostly involve passing Python integers, floats (=C doubles), strings, and NumPy 1D and 2D float and integer arrays. I rarely need to pass anything else to a C routine to do a calculation. If you are in the same situation as me, then this package I put together might help you. It turns out to be fairly easy once you get going.

Please note, Tom Loredo is not responsible for any errors in my code or instructions although I am deeply indebted to him. Likewise, this code is for research only. It was tested by only my development and usage. It is not guaranteed, and comes with no warranty. Do not use this code where there are any threats of loss of life, limb, property, or money or anything you or others hold dear.

I developed these C extensions and their Python wrappers on a Macintosh G4 laptop using system OS X 10.4 (essential BSD Unix), Python 2.4, NumPy 0.9x, and the gnu compiler and linker gcc. I think most of what I tell you here will be easily translated to Linux and other Unix systems beyond the Mac. I am not sure about Windows. I hope that my low-level approach will make it easy for Windows users, too.

The code (both C and Python) for the extensions may look like a lot, but it is *very* repetitious. Once you get the main scheme for one extension function you will see that repeated over and over again in all the others with minor variations to handle different arguments or return different objects to the calling routine. Don’t be put off by the code. The good news is that for many numerical uses extensions will follow the same format so you can quickly reuse what you already have written for new projects. Focus on one extension function and follow it in detail (in fact, I will do this below). Once you understand it, the other routines will be almost obvious. The same is true of the several utility functions that come with the package. They help you create, test, and manipulate the data and they also have a lot of repetition. The utility functions are also very short and simple so nothing to fear there.

General Scheme for NumPy Extensions

This will be covered in detail below, but first I wanted to give you a sense of how each extension is organized.

Three things that must be done before your C extension functions in the C source file.

1. You must include Python and NumPy headers.
2. Each extension must be named in a defining structure at the beginning of the file. This is a name used to access the extension from a Python function.
3. Next an initialization set of calls is made to set up the Python and NumPy calls and interface. It will be the same for all extensions involving NumPy and Python unless you add extensions to access other Python packages or classes beyond NumPy arrays. I will not cover any of that here (because I don't know it). So the init calls can be copied to each extension file.

Each C extension will have the following form.

- The arguments will always be the same: (`PyObject *self` , `PyObject *args`) - Don't worry if you don't know what exactly these are. They are pointers to general Python objects and they are automatically provided by the header files you will use from NumPy and Python itself. You need know no more than that.
- The args get processed by a function call that parses them and assigns them to C defined objects.
- Next the results of that parse might be checked by a utility routine that reaches into the structure representing the object and makes sure the data is the right kind (float or int, 1D or 2D array, etc.). Although I included some of these C-level checks, you will see that I think they are better done in Python functions that are used to wrap the C extensions. They are also a lot easier to do in Python. I have plenty of data checks in my calling Python wrappers. Usually this does not lead to much overhead since you are not calling these extensions billions of times in some loop, but using them as a portal to a C or C++ routine to do a long, complex, repetitive, specialized calculation.
- After some possible data checks, C data types are initialized to point to the data part of the NumPy arrays with the help of utility functions.
- Next dimension information is extracted so you know the number of columns, rows, vector dimensions, etc.
- Now you can use the C arrays to manipulate the data in the NumPy arrays. The C arrays and C data from the above parse point to the original Python/NumPy data so changes you make affect the array values when you go back to Python after the extension returns. You can pass the arrays to other C functions that do calculations, etc. Just remember you are operating on the original NumPy matrices and vectors.
- After your calculation you have to free any memory allocated in the construction of your C data for the NumPy arrays. This is done again by Utility functions. This step is only necessary if you allocated memory to handle the arrays (e.g. in the matrix routines), but is not necessary if you have not allocated memory (e.g. in the vector routines).
- Finally, you return to the Python calling function, by returning a Python value or NumPy array. I have C extensions which show examples of both.

Python Wrapper Functions

It is best to call the C extensions by calling a Python function that then calls the extension. This is called a Python wrapper function. It puts a more pythonic look to your code (e.g. you can use keywords easily) and, as I pointed out above, allows you to easily check that the function arguments and data are correct before you had them over to the C extension and other C functions for that big calculation. It may seem like an unnecessary extra step, but it’s worth it.

The Code

In this section I refer to the code in the source files `C_arraytest.h` , `C_arraytest.c` , `C_arraytest.py` , and `C_arraytest.mak` . You should keep those files handy (probably printed out) so you can follow the explanations of the code below.

The C Code – one detailed example with utilities

First, I will use the example of code from C*arraytest.h, C_arraytest.c for the routine called matsq. This function takes a (NumPy) matrix _A*, integer *i*, and (Python) float *y* as input and outputs a return (NumPy) matrix *B* each of whose components is equal to the square of the input matrix component times the integer times the float. Mathematically:

$$B_{ij} = i y (A_{ij})^2$$

The Python code to call the matsq routine is `A=matsq(B,i,y)` . Here is the relevant code in one place:

The Header file, C_arraytest.h:

```
/* Header to test of C modules for arrays for Python: C_test.c */

/* ==== Prototypes =================================== */

// .... Python callable Vector functions ..................
static PyObject *vecfcn1(PyObject *self, PyObject *args);
static PyObject *vecsq(PyObject *self, PyObject *args);

/* .... C vector utility functions ..................*/
PyArrayObject *pyvector(PyObject *objin);
double *pyvector_to_Carrayptrs(PyArrayObject *arrayin);
int  not_doublevector(PyArrayObject *vec);

/* .... Python callable Matrix functions ..................*/
static PyObject *rowx2(PyObject *self, PyObject *args);
static PyObject *rowx2_v2(PyObject *self, PyObject *args);
static PyObject *matsq(PyObject *self, PyObject *args);
static PyObject *contigmat(PyObject *self, PyObject *args);

/* .... C matrix utility functions ..................*/
PyArrayObject *pymatrix(PyObject *objin);
double **pymatrix_to_Carrayptrs(PyArrayObject *arrayin);
double **ptrvector(long n);
void free_Carrayptrs(double **v);
int  not_doublematrix(PyArrayObject *mat);

/* .... Python callable integer 2D array functions ...............
static PyObject *intfcn1(PyObject *self, PyObject *args);

/* .... C 2D int array utility functions ..................*/
PyArrayObject *pyint2Darray(PyObject *objin);
int **pyint2Darray_to_Carrayptrs(PyArrayObject *arrayin);
int **ptrintvector(long n);
void free_Cint2Darrayptrs(int **v);
int  not_int2Darray(PyArrayObject *mat);
```

The Source file, C_arraytest.c:

```
/* A file to test imorting C modules for handling arrays to Python

#include "Python.h"
#include "arrayobject.h"
#include "C_arraytest.h"
#include

/* #### Globals #################################### */

/* ==== Set up the methods table ====================== */
static PyMethodDef _C_arraytestMethods[] = {
    {"vecfcn1", vecfcn1, METH_VARARGS},
```

```
    {"vecsq", vecsq, METH_VARARGS},
    {"rowx2", rowx2, METH_VARARGS},
    {"rowx2_v2", rowx2_v2, METH_VARARGS},
    {"matsq", matsq, METH_VARARGS},
    {"contigmat", contigmat, METH_VARARGS},
    {"intfcn1", intfcn1, METH_VARARGS},
    {NULL, NULL}     /* Sentinel - marks the end of this structure
};

/* ==== Initialize the C_test functions ====================== */
// Module name must be _C_arraytest in compile and linked
void init_C_arraytest()  {
    (void) Py_InitModule("_C_arraytest", _C_arraytestMethods);
    import_array();  // Must be present for NumPy.  Called first af
}

/* #### Vector Extensions ############################## */

/* ==== vector function - manipulate vector in place ==============
    Multiply the input by 2 x dfac and put in output
    Interface:  vecfcn1(vec1, vec2, str1, d1)
                vec1, vec2 are NumPy vectors,
                str1 is Python string, d1 is Python float (double)
                Returns integer 1 if successful                */
static PyObject *vecfcn1(PyObject *self, PyObject *args)
{
    PyArrayObject *vecin, *vecout;  // The python objects to be ext
    double *cin, *cout;             // The C vectors to be created
                                    //   python vectors, cin and co
                                    //   of vecin and vecout, respe
    int i,j,n;
    const char *str;
    double dfac;

    /* Parse tuples separately since args will differ between C fcn
    if (!PyArg_ParseTuple(args, "O!O!sd", &PyArray_Type, &vecin,
        &PyArray_Type, &vecout, &str, &dfac))  return NULL;
    if (NULL == vecin)  return NULL;
    if (NULL == vecout)  return NULL;

    // Print out input string
    printf("Input string: %s\n", str);

    /* Check that objects are 'double' type and vectors
         Not needed if python wrapper function checks before call t
    if (not_doublevector(vecin)) return NULL;
    if (not_doublevector(vecout)) return NULL;

    /* Change contiguous arrays into C * arrays   */
    cin=pyvector_to_Carrayptrs(vecin);
    cout=pyvector_to_Carrayptrs(vecout);

    /* Get vector dimension. */
```

```
    n=vecin->dimensions[0];

    /* Operate on the vectors  */
    for ( i=0; i<n; i++)="" {="" cout[i]="2.0*dfac*cin[i];" }="" re

    /* Make a new double vector of same dimension */
    vecout=(PyArrayObject *) PyArray_FromDims(1,dims,NPY_DOUBLE);

    /* Change contiguous arrays into C *arrays   */
    cin=pyvector_to_Carrayptrs(vecin);
    cout=pyvector_to_Carrayptrs(vecout);

    /* Do the calculation. */
    for ( i=0; i<n; i++)="" {="" cout[i]="dfactor*cin[i]*cin[i];" }
    return (double *) arrayin->data;  /* pointer to arrayin data as
}
/* ==== Check that PyArrayObject is a double (Float) type and a vec
    return 1 if an error and raise exception */
int  not_doublevector(PyArrayObject *vec)  {
    if (vec->descr->type_num != NPY_DOUBLE || vec->nd != 1)  {
        PyErr_SetString(PyExc_ValueError,
            "In not_doublevector: array must be of type Float and 1
        return 1;  }
    return 0;
}

/* #### Matrix Extensions ############################## */

/* ==== Row x 2 function - manipulate matrix in place ============
    Multiply the 2nd row of the input by 2 and put in output
    interface:  rowx2(mat1, mat2)
                mat1 and mat2 are NumPy matrices
                Returns integer 1 if successful
static PyObject *rowx2(PyObject *self, PyObject *args)
{
    PyArrayObject *matin, *matout;  // The python objects to be ext
    double **cin, **cout;           // The C matrices to be created
                                    //   python matrices, cin and c
                                    //   of matin and matout, respe
    int i,j,n,m;

    /* Parse tuples separately since args will differ between C fcn
    if (!PyArg_ParseTuple(args, "O!O!", &PyArray_Type, &matin,
        &PyArray_Type, &matout))  return NULL;
    if (NULL == matin)  return NULL;
    if (NULL == matout)  return NULL;

    /* Check that objects are 'double' type and matrices
         Not needed if python wrapper function checks before call t
    if (not_doublematrix(matin)) return NULL;
    if (not_doublematrix(matout)) return NULL;

    /* Change contiguous arrays into C ** arrays (Memory is Allocat
```

```
    cin=pymatrix_to_Carrayptrs(matin);
    cout=pymatrix_to_Carrayptrs(matout);

    /* Get matrix dimensions. */
    n=matin->dimensions[0];
    m=matin->dimensions[1];

    /* Operate on the matrices  */
    for ( i=0; i<n; 1="" 2="" i++)="" {="" for="" (="" j="0;" j<m;=
    m=matin->dimensions[1];

    /* Operate on the matrices  */
    for ( i=0; i<n; i++)="" {="" for="" (="" j="0;" j<m;="" j++)=""
    m=dims[1]=matin->dimensions[1];

    /* Make a new double matrix of same dims */
    matout=(PyArrayObject *) PyArray_FromDims(2,dims,NPY_DOUBLE);

    /* Change contiguous arrays into C ** arrays (Memory is Allocat
    cin=pymatrix_to_Carrayptrs(matin);
    cout=pymatrix_to_Carrayptrs(matout);

    /* Do the calculation. */
    for ( i=0; i<n; i++)="" {="" for="" (="" j="0;" j<m;="" j++)=""
    m=dims[1]=matin->dimensions[1];
    ncomps=n*m;

    /* Make a new double matrix of same dims */
    matout=(PyArrayObject *) PyArray_FromDims(2,dims,NPY_DOUBLE);

    /* Change contiguous arrays into C * arrays pointers to PyArray
    cin=pyvector_to_Carrayptrs(matin);
    cout=pyvector_to_Carrayptrs(matout);

    /* Do the calculation. */
    printf("In contigmat, cout (as contiguous memory) =\n");
    for ( i=0; i<ncomps; i++)="" {="" cout[i]="cin[i]-x1;" printf('
    m=arrayin->dimensions[1];
    c=ptrvector(n);
    a=(double *) arrayin->data;  /* pointer to arrayin data as doub
    for ( i=0; i<n; 1="" i++)="" {="" c[i]="a+i*m;" }="" return=""
        PyErr_SetString(PyExc_ValueError,
            "In not_doublematrix: array must be of type Float and 2
        return 1;  }
    return 0;
}

/* #### Integer 2D Array Extensions ##############################

/* ==== Integer function - manipulate integer 2D array in place ===
    Replace >=0 integer with 1 and < 0 integer with 0 and put in ou
    interface:  intfcn1(int1, afloat)
                int1 is a NumPy integer 2D array, afloat is a Pytho
```

```
                Returns integer 1 if successful
static PyObject *intfcn1(PyObject *self, PyObject *args)
{
    PyArrayObject *intin, *intout;  // The python objects to be ext
    int **cin, **cout;              // The C integer 2D arrays to b
                                    //   python integer 2D arrays,
                                    //   of intin and intout, respe
    int i,j,n,m, dims[2];
    double afloat;

    /* Parse tuples separately since args will differ between C fcn
    if (!PyArg_ParseTuple(args, "O!d",
        &PyArray_Type, &intin, &afloat))  return NULL;
    if (NULL == intin)  return NULL;

    printf("In intfcn1, the input Python float = %e, a C double\n",

    /* Check that object input is int type and a 2D array
       Not needed if python wrapper function checks before call to
    if (not_int2Darray(intin)) return NULL;

    /* Get the dimensions of the input */
    n=dims[0]=intin->dimensions[0];
    m=dims[1]=intin->dimensions[1];

    /* Make a new int array of same dims */
    intout=(PyArrayObject *) PyArray_FromDims(2,dims,NPY_LONG);

    /* Change contiguous arrays into C ** arrays (Memory is Allocat
    cin=pyint2Darray_to_Carrayptrs(intin);
    cout=pyint2Darray_to_Carrayptrs(intout);

    /* Do the calculation. */
    for ( i=0; i<n; i++)="" {="" for="" (="" j="0;" j<m;="" j++)=""
                cout[i][j]= 1;  }
            else  {
                cout[i][j]= 0;  }
    }  }

    printf("In intfcn1, the output array is,\n\n");

    for ( i=0; i<n; i++)="" {="" for="" (="" j="0;" j<m;="" j++)=""
    m=arrayin->dimensions[1];
    c=ptrintvector(n);
    a=(int *) arrayin->data;  /* pointer to arrayin data as int */
    for ( i=0; i<n; 1="" i++)="" {="" c[i]="a+i*m;" }="" return=""
        PyErr_SetString(PyExc_ValueError,
            "In not_int2Darray: array must be of type int and 2 dir
        return 1;  }
    return 0;
}

// EOF</n;> </n;></n;></n;></ncomps;></n;></n;></n;></n;></n;>
```



Now, lets look at the source code in smaller chunks.

Headers

You must include the following headers with Python.h **always** the first header included.

```
#include "Python.h"
#include "arrayobject.h"
```

I also include the header C_arraytest.h which contains the prototype of the matsq function:

```
static PyObject *matsq(PyObject *self, PyObject *args);
```

The static keyword in front of a function declaration makes this function private to your extension module. The linker just won't see it. This way you can use the same intuitional function names(i.e. sum, check, trace) for all extension modules without having name clashes between them at link time. The type of the function is `PyObject *` because it will always be returning to a Python calling function so you can (must, actually) return a Python object. The arguments are always the same,

```
PyObject *self and PyObject *args
```

The first one self is never used, but necessary because of how Python passes arguments. The second args is a pointer to a Python tuple that contains all of the arguments (B,i,x) of the function.

Method definitions

This sets up a table of function names that will be the interface from your Python code to your C extension. The name of the C extension module will be `_C_arraytest` (note the leading underscore). It is important to get the name right each time it is used because there are strict requirements on using the module name in the code. The name appears first in the method definitions table as the first part of the table name:

```
static PyMethodDef _C_arraytestMethods[] = {
  ...,
  {"matsq", matsq, METH_VARARGS},
  ...,
  {NULL, NULL}     /* Sentinel - marks the end of this structure */
};
```

where I used ellipses (...) to ignore other code not relevant to this function. The `METH_VARARGS` parameter tells the compiler that you will pass the arguments the usual way without keywords as in the example `A=matsq(B,i,x)` above. There are ways to use Python keywords, but I have not tried them out. The table should always end with {NULL, NULL} which is just a “marker” to note the end of the table.

Initializations

These functions tell the Python interpreter what to call when the module is loaded. Note the name of the module (`_C_arraytest`) must come directly after the init in the name of the initialization structure.

```
void init_C_arraytest() {
  (void) Py_InitModule("_C_arraytest", _C_arraytestMethods);
  import_array(); // Must be present for NumPy. Called first after
}
```

The order is important and you must call these two initialization functions first.

The matsqfunction code

Now here is the actual function that you will call from Python code. I will split it up and explain each section.

The function name and type:

```
static PyObject *matsq(PyObject *self, PyObject *args)
```

You can see they match the prototype in C_arraytest.h.

The local variables:

```
PyArrayObject *matin, *matout;
double **cin, **cout, dfactor;
int i,j,n,m, dims[2], ifactor;
```

The `PyArrayObjects` are structures defined in the NumPy header file and they will be assigned pointers to the actual input and output NumPy arrays (A and B). The C arrays `cin` and `cout` are Cpointers that will point (eventually) to the actual data in the NumPy arrays and allow you to manipulate it. The variable `dfactor` will be the Python float y, `ifactor` will be the Python int i, the variables i,j,n, and m will be loop variables (i and j) and matrix dimensions (n= number of rows, m= number of columns) in A and B. The array dims will be used to access n and m from the NumPy array. All this happens below. First we have to extract the input variables (A, i, y) from the args tuple. This is done by the call,

```
/* Parse tuples separately since args will differ between C fcns */
if (!PyArg_ParseTuple(args, "O!id",
    &PyArray_Type, &matin, &ifactor, &dfactor)) return NULL;
```

The `PyArg_ParseTuple` function takes the args tuple and using the format string that appears next (“O!id”) it assigns each member of the tuple to a C variable. Note you must pass all C variables by reference. This is true even if the C variable is a pointer to a string (see code in vecfcn1 routine). The format string tells the parsing function what type of variable to use. The common variables for Python all have letter names (e.g. s for string, i for integer, d for (double - the Python float)). You can find a list of these and many more in Guido’s tutorial (http://docs.python.org/ext/ext.html). For data types that are not in standard Python like the NumPy arrays you use the O! notation which tells the parser to look for a type structure (in this case a NumPy structure `PyArray_Type`) to help it convert the tuple member that will be assigned to the local variable (matin) pointing to the NumPy array structure. Note these are also passed by reference. The order must be maintained and match the calling interface of the Python function you want. The format string defines the interface and if you do not call the function from Python so the number of arguments match the number in the format string, you will get an error. This is good since it will point to where the problem is.

If this doesn’t work we return NULL which will cause a Python exception.

```
if (NULL == matin) return NULL;
```

Next we have a check that the input matrix really is a matrix of NumPy type double. This test is also done in the Python wrapper for this C extension. It is better to do it there, but I include the test here to show you that you can do testing in the C extension and you can “reach into” the NumPy structure to pick out it’s parameters. The utility function `not_doublematrix` is explained later.

```
/* Check that object input is 'double' type and a matrix
   Not needed if python wrapper function checks before call to this
if (not_doublematrix(matin)) return NULL;
```

Here's an example of reaching into the NumPy structure to get the dimensions of the matrix matin and assign them to local variables as mentioned above.

```
/* Get the dimensions of the input */
n=dims[0]=matin->dimensions[0];
m=dims[1]=matin->dimensions[1];
```

Now we use these matrix parameters to generate a new NumPy matrix matout (our output) right here in our C extension. PyArray_FromDims(2,dims,NPY_DOUBLE) is a utility function provided by NumPy (not me) and its arguments tell NumPy the rank of the NumPy object (2), the size of each dimension (dims), and the data type (NPY_DOUBLE). Other examples of creating different NumPy arrays are in the other C extensions.

```
/* Make a new double matrix of same dims */
matout=(PyArrayObject *) PyArray_FromDims(2,dims,NPY_DOUBLE);
```

To actually do our calculations we need C structures to handle our data so we generate two C 2-dimensional arrays (cin and cout) which will point to the data in matin and matout, respectively. Note, here memory is allocated since we need to create an array of pointers to C doubles so we can address cin and cout like usual C matrices with two indices. This memory must be released at the end of this C extension. Memory allocation like this is not always necessary. See the routines for NumPy vector manipulation and treating NumPy matrices like contiguous arrays (as they are in NumPy) in the C extension (the routine contigmat).

```
/* Change contiguous arrays into C ** arrays (Memory is Allocated!)
cin=pymatrix_to_Carrayptrs(matin);
cout=pymatrix_to_Carrayptrs(matout);
```

Finally, we get to the point where we can manipulate the matrices and do our calculations. Here is the part where the original equation operations `\(B_{ij}= i y (A_{ij})^2\)` are carried out. Note, we are directly manipulating the data in the original NumPy arrays A and B passed to this extension. So anything you do here to the components of cin or cout will be done to the original matrices and will appear there when you return to the Python code.

```
/* Do the calculation. */
for ( i=0; i<n; i++) {
   for ( j=0; j<m; j++) {
      cout[i][j]= ifactor*dfactor*cin[i][j]*cin[i][j];
   }
}
```

We are ready to go back to the Python calling routine, but first we release the memory we allocated for cin and cout.

```
/* Free memory, close file and return */
free_Carrayptrs(cin);
free_Carrayptrs(cout);
```

Now we return the result of the calculation.

```
return PyArray_Return(matout);
```

If you look at the other C extensions you can see that you can also return regular Python variables (like ints) using another Python-provided function `Py_BuildValue("i", 1)` where the string “i” tells the function the data type and the second argument (1 here) is the data value to be returned. If you decide to return nothing, you ‘‘must‘’ return the Python keyword None like this:

```
Py_INCREF(Py_None);
return Py_None;
```

The Py_INCREF function increases the number of references to None (remember Python collects allocated memory automatically when there are no more references to the data). You must be careful about this in the C extensions. For more info see Guido´s tutorial.

The utility functions

Here are some quick descriptions of the matrix utility functions. They are pretty much self-explanatory. The vector and integer array utility functions are very similar.

The first utility function is not used in any of the C extensions here, but I include it because a helpful person sent it along with some code and it does show how one might convert a python object to a NumPy array. I have not tried it. Use at your own risk.

```
PyArrayObject *pymatrix(PyObject *objin) {
   return (PyArrayObject *) PyArray_ContiguousFromObject(objin,
      NPY_DOUBLE, 2,2);
}
```

The next one creates the C arrays that are used to point to the rows of the NumPy matrices. This allocates arrays of pointers which point into the NumPy data. The NumPy data is contiguous and strides (m) are used to access each row. This function calls ptrvector(n) which does the actual memory allocation. Remember to deallocate memory after using this one.

```
double **pymatrix_to_Carrayptrs(PyArrayObject *arrayin) {
   double **c, *a;
   int i,n,m;

   n=arrayin->dimensions[0];
   m=arrayin->dimensions[1];
   c=ptrvector(n);
   a=(double *) arrayin->data; /* pointer to arrayin data as double
   for ( i=0; i<n; i++) {
      c[i]=a+i*m;
   }
   return c;
}
```

Here is where the memory for the C arrays of pointers is allocated. It's a pretty standard memory allocator for arrays.

```
double **ptrvector(long n) {
   double **v;
   v=(double **)malloc((size_t) (n*sizeof(double)));
   if (!v)   {
      printf("In **ptrvector. Allocation of memory for double array
      exit(0);
   }
   return v;
}
```

This is the routine to deallocate the memory.

```
 void free_Carrayptrs(double **v) {
   free((char*) v);
}
```

Note: There is a standard C-API for converting from Python objects to C-style arrays-of-pointers called PyArray_AsCArray

Here is a utility function that checks to make sure the object produced by the parse is a NumPy matrix. You can see how it reaches into the NumPy object structure.

```
int not_doublematrix(PyArrayObject *mat) {
   if (mat->descr->type_num != NPY_DOUBLE || mat->nd != 2) {
      PyErr_SetString(PyExc_ValueError,
         "In not_doublematrix: array must be of type Float and 2 di
      return 1; }
   return 0;
}
```

The C Code – other variations

As I mentioned in the introduction the functions are repetitious. All the other functions follow a very similar pattern. They are given a line in the methods structure, they have the same arguments, they parse the arguments, they may check the C structures after the parsing, they set up C variables to manipulate which point to the input data, they do the actual calculation, they deallocate memory (if necessary) and they return something for Python (either None or a Python object). I'll just mention some of the differences in the code from the above matrix C extension matsq.

vecfcn1:

The format string for the parse function specifies that a variable from Python is a string (s).

```
if (!PyArg_ParseTuple(args, "O!O!sd", &PyArray_Type, &vecin,
      &PyArray_Type, &vecout, &str, &dfac)) return NULL;
```

No memory is allocated in the pointer assignments for the local C arrays because they are already vectors.

```
cin=pyvector_to_Carrayptrs(vecin);
cout=pyvector_to_Carrayptrs(vecout);
```

The return is an int = 1 if successful. This is returned as a Python int.

```
return Py_BuildValue("i", 1);
```

rowx2:

In this routine we “pass back” the output using the fact that it is passed by reference in the argument tuple list and is changed in place by the manipulations. Compare this to returning an array from the C extension in matsq. Either one gets the job done.

contigmat:

Here the matrix data is treated like a long vector (just like stacking the rows end to end). This is useful if you have array classes in C++ which store the data as one long vector and then use strides to access it like an array (two-dimensional, three-dimensional, or whatever). Even though matin and matout are “matrices” we treat them like vectors and use the vector utilities to get our C pointers cin and cout.

```
/* Change contiguous arrays into C * arrays pointers to PyArrayObje
cin=pyvector_to_Carrayptrs(matin);
cout=pyvector_to_Carrayptrs(matout);
```

For other utility functions note that we use different rank, dimensions, and NumPy parameters (e.g. `NPY_LONG`) to tell the routines we are calling what the data types are.

The Make File

The make file is very simple. It is written for Mac OS X 10.4, as BSD Unix.

```
# ---- Link ---------------------------
_C_arraytest.so: C_arraytest.o C_arraytest.mak      gcc -bundle -fl

# ---- gcc C compile ------------------
C_arraytest.o: C_arraytest.c C_arraytest.h C_arraytest.mak      gcc

-I/Library/Frameworks/Python.framework/Versions/2.4/include/python2
-I/Library/Frameworks/Python.framework/Versions/2.4/lib/python2.4/s
```

The compile step is pretty standard. You do need to add paths to the Python headers:

`-I/Library/Frameworks/Python.framework/Versions/2.4/include/python2`

and paths to NumPy headers:

`-I/Library/Frameworks/Python.framework/Versions/2.4/lib/python2.4/si`

These paths are for a Framework Python 2.4 install on Mac OS X. You need to supply paths to the headers installed on your computer. They ma y be different. My guess is the gcc flags will be the same for the compile.

The link step produces the actual module (`_C_arraytest.so`) which can be imported to Python code. This is specific to the Mac OS X system. On Linux or Windows you will need something different. I have been searching for generic examples, but I'm not sure what I found would w ork for most people so I chose not to display the findings there. I cannot judge whether the code is good for those systems.

Note, again the name of the produced shared library '''must''' match the name in the initialization and methods definition calls in the C extension source code. Hence the leading underline in the name `_C_arraytest.so` .

Here's my modified Makefile which compiles this code under Linux (save it as Makefile in the same directory, and then run 'make'

– PaulIvanov

```
# ---- Link ---------------------------
_C_arraytest.so:  C_arraytest.o
        gcc -shared C_arraytest.o -o _C_arraytest.so

# ---- gcc C compile ------------------
C_arraytest.o:  C_arraytest.c C_arraytest.h
        gcc  -c C_arraytest.c -I/usr/include/python2.4 -I/usr/lib/p
```

The Python Wrapper Code

Here as in the C code I will only show detailed description of one wrapper function and its use. There is so much repetition that the other wrappers will be clear if you understand one. I will again use the matsq function. This is the code that will first be called when you inv oke the matsq function in your own Python code after importing the wrapper module (`C_arraytest.py`) which automatically imports and uses (in a way hidden from the user) the actual C extensions in `_C_arraytest.so` (note the leading underscore which keeps the names separate).

imports:

Import the C extensions, NumPy, and the system module (used for the exit statement at the end which is optional).

```
import _C_arraytest
import numpy as NP
import sys
```

The definition of the Python matsq function

Pass a NumPy matrix (matin), a Python int (ifac), and a Python float (dfac). Check the arguments to make sure they are the right type and dimensions and size. This is much easier and safer on the Python side which is why I do it here even though I showed a way to do some of this in the C extensions.

```
def matsq(matin, ifac, dfac):
    # .... Check arguments, double NumPy matrices?
    test=NP.zeros((2,2)) # create a NumPy matrix as a test object t
    typetest= type(test) # get its type
    datatest=test.dtype   # get data type
    if type(matin) != typetest:
        raise 'In matsq, matrix argument is not *NumPy* array'
    if len(NP.shape(matin)) != 2:
        raise 'In matsq, matrix argument is not NumPy *matrix*'
    if matin.dtype != datatest:
        raise 'In matsq, matrix argument is not *Float* NumPy matri
    if type(ifac) != type(1):
        raise 'In matsq, ifac argument is not an int'
    if type(dfac) != type(1.0):
        raise 'In matsq, dfac argument is not a python float'
```

Finally, call the C extension to do the actual calculation on matin.

```
# .... Call C extension function
return _C_arraytest.matsq(matin, ifac, dfac)
```

You can see that the python part is the easiest.

Using your C extension

If the test function `mattest2` were in another module (one you were writing), here’s how you would use it to call the wrapped matsq function in a script.

```
from C_arraytest import * # Note, NO underscore since you are impor

import numpy as NP
import sys

def mattest2():
    print "\n--- Test matsq ------------------------------"
    print " Each 2nd matrix component should = square of 1st matri
    n=4 # Number of columns      # Make 2 x n matrices
    z=NP.arange(float(n))
    x=NP.array([z,3.0*z])
    jfac=2
    xfac=1.5
    y=matsq(x, jfac, xfac)
    print "x=",x
    print "y=",y

if __name__ == '__main__':
    mattest2()
```

The output looks like this:

```
--- Test matsq ------------------------------
Each 2nd matrix component should = square of 1st matrix component x

x= [[ 0\. 1\. 2\. 3.]
 [ 0\. 3\. 6\. 9.]]
y= [[   0\.    3\.   12\.   27.]
 [   0\.   27\. 108\. 243.]]
```

The output of all the test functions is in the file `C_arraytest` output.

Summary

This is the first draft explaining the C extensions I wrote (with help). If you have comments on the code, mistakes, etc. Please post them on the pythonmac email list. I will see them.

I wrote the C extensions to work with NumPy although they were originally written for Numeric. If you must use Numeric you should test them to see if they are compatible. I suspect names like `NPY_DOUBLE`, for example, will not be. I strongly suggest you upgrade to the NumPy since it is the future of Numeric in Python. It's worth the effort.

Comments?!

Note that this line, while in the header file above, is missing from the .h in the tar-ball.

```
static PyObject *rowx2_v2(PyObject *self, PyObject *args);
```

– PaulIvanov

The output file name should be _C_arraytest_d.pyd for Debug version and _C_arraytest.pyd for Release version.

– Geoffrey Zhu

---

ptrvector() allocates `n*sizeof(double)` , but should really allocate pointers to double; so: `n*sizeof(double *)`

– Peter Meerwald

In vecsq(), line 86, since you will be creating a 1-dimensional vector:

```
int i,j,n,m, dims[2];
```

should be:

```
int i,j,n,m, dims[1];
```

In `pyvector_to_Carrayptrs()` , `n` is never used.

– FredSpiessens

Attachments

- `C_arraytest.c`
- `C_arraytest.c_v2`
- `C_arraytest.h`
- `C_arraytest.h_v2`
- `Cext_v2.tar.gz`

Embedding in Traits GUI

Embedding a Matplotlib Figure in a Traits App

Traits, part of the Enthought Tools Suit, provides a great framework for creating GUI Apps without a lot of the normal boilerplate required to connect the UI the rest of the application logic. A brief introduction to Traits can be found here. Although ETS comes with it's own traits-aware plotting framework (Chaco), if you already know matplotlib it is just as easy to embed this instead. The advantages of Chaco (IMHO) are its interactive "tools", an (in development) OpenGL rendering backend and an easy-to-understand codebase. However, matplotlib has more and better documentation and better defaults; it just works. The key to getting TraitsUI and matplotlib to play nice is to use the mpl object-oriented API, rather than pylab / pyplot. This recipe requires the following packages:

- numpy
- wxPython
- matplotlib
- Traits > 3.0
- TraitsGUI > 3.0
- TraitsBackendWX > 3.0

For this example, we will display a function (y, a sine wave) of one variable (x, a numpy ndarray) and one parameter (scale, a float value with bounds). We want to be able to vary the parameter from the UI and see the resulting changes to y in a plot window. Here's what the final result looks like: The TraitsUI "!CustomEditor" can be used to display any wxPython window as the editor for the object. You simply pass the !CustomEditor a callable which, when called, returns the wxPython window you want to display. In this case, our !MakePlot() function returns a wxPanel containing the mpl !FigureCanvas and Navigation toolbar. This example exploits a few of Traits' features. We use "dynamic initialisation" to create the Axes and Line2D objects on demand (using the _xxx_default methods). We use Traits "notification", to call update_line(...) whenever the x- or y-data is changed. Further, the y-data is declared as a Property trait which depends on both the 'scale' parameter and the x-data. 'y' is then recalculated on demand, whenever either 'scale' or 'x' change. The 'cached_property' decorator prevents recalculation of y if it's dependancies `*` are `)#` not `*` modified.`

Finally, there's a bit of wx-magic in the redraw() method to limit the redraw rate by delaying the actual drawing by 50ms. This uses the wx.!CallLater class. This prevents excessive redrawing as the slider is dragged, keeping the UI from lagging. `Here's` the full listing:

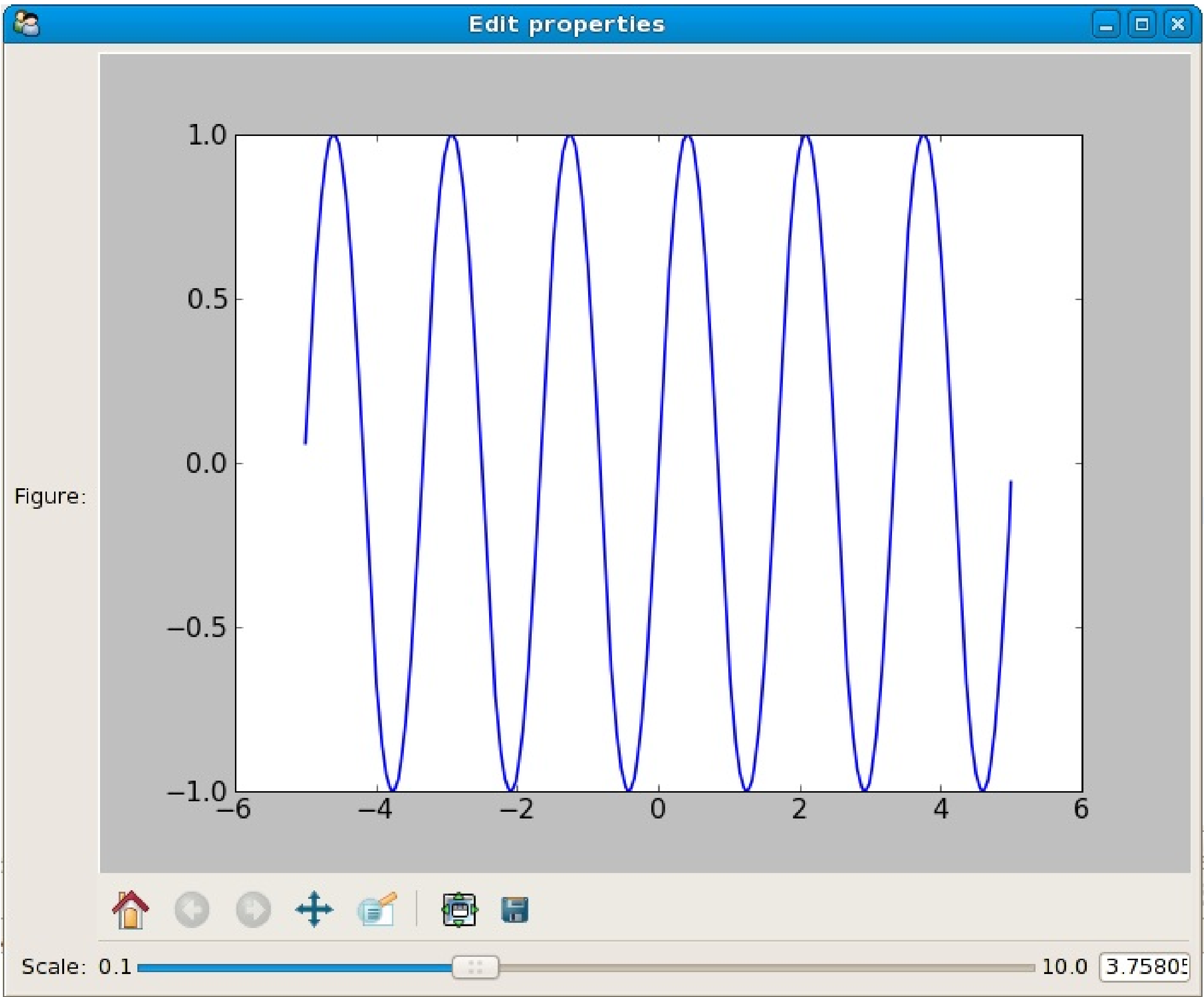
```
#!python
"""
```

```
A simple demonstration of embedding a matplotlib plot window in
a traits-application. The CustomEditor allow any wxPython window
to be used as an editor. The demo also illustrates Property traits,
which provide nice dependency-handling and dynamic initialisation,
the _xxx_default(...) method.
"""
from enthought.traits.api import HasTraits, Instance, Range,\
                                Array, on_trait_change, Property,\
                                cached_property, Bool
from enthought.traits.ui.api import View, Item
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg
from matplotlib.backends.backend_wx import NavigationToolbar2Wx
from matplotlib.figure import Figure
from matplotlib.axes import Axes
from matplotlib.lines import Line2D
from enthought.traits.ui.api import CustomEditor
import wx
import numpy
def MakePlot(parent, editor):
    """
 Builds the Canvas window for displaying the mpl-figure
 """
    fig = editor.object.figure
    panel = wx.Panel(parent, -1)
    canvas = FigureCanvasWxAgg(panel, -1, fig)
    toolbar = NavigationToolbar2Wx(canvas)
    toolbar.Realize()
    sizer = wx.BoxSizer(wx.VERTICAL)
    sizer.Add(canvas,1,wx.EXPAND|wx.ALL,1)
    sizer.Add(toolbar,0,wx.EXPAND|wx.ALL,1)
    panel.SetSizer(sizer)
    return panel
class PlotModel(HasTraits):
    """A Model for displaying a matplotlib figure"""
    #we need instances of a Figure, a Axes and a Line2D
    figure = Instance(Figure, ())
    axes = Instance(Axes)
    line = Instance(Line2D)
    _draw_pending = Bool(False) #a flag to throttle the redraw rate
    #a variable paremeter
    scale = Range(0.1,10.0)
    #an independent variable
    x = Array(value=numpy.linspace(-5,5,512))
    #a dependent variable
    y = Property(Array, depends_on=['scale','x'])
    traits_view = View(
                    Item('figure',
                         editor=CustomEditor(MakePlot),
                         resizable=True),
                    Item('scale'),
                    resizable=True
                    )
    def _axes_default(self):
```

```
            return self.figure.add_subplot(111)
        def _line_default(self):
            return self.axes.plot(self.x, self.y)[0]
        @cached_property
        def _get_y(self):
            return numpy.sin(self.scale * self.x)
        @on_trait_change("x, y")
        def update_line(self, obj, name, val):
            attr = {'x': "set_xdata", 'y': "set_ydata"}[name]
            getattr(self.line, attr)(val)
            self.redraw()
        def redraw(self):
            if self._draw_pending:
                return
            canvas = self.figure.canvas
            if canvas is None:
                return
            def _draw():
                canvas.draw()
                self._draw_pending = False
            wx.CallLater(50, _draw).Start()
            self._draw_pending = True
    if __name__=="__main__":
        model = PlotModel(scale=2.0)
        model.configure_traits()
```

Attachments

- `mpl_editor.py`
- `mpl_in_traits_view.png`


Edit properties
Figure:
1.0
0.5
0.0
−0.5
−1.0
−6
−4
−2
0
2
4
6
Scale: 0.1
10.0
3.75805

Matplotlib: drag'n'drop text example

introduction

Matplotlib provides event handling to determine things like key presses, mouse position, and button clicks. Matplotlib supports a number of GUIs, and provides an interface to the GUI event handling via the mpl_connect and mpl_disconnect methods.

This page gives an example of use of these facilities by adding a Drag'n'Drop handler for text objects. You can get the source code for this example here: `Text_DragnDrop_v0.1.py` .)

Defining the handler class

```
#!python numbers=disable
from matplotlib         import pylab as p
from matplotlib.text        import Text

class DragHandler(object):
    """ A simple class to handle Drag n Drop.

 This is a simple example, which works for Text objects only
 """
    def __init__(self, figure=None) :
        """ Create a new drag handler and connect it to the figure
 If the figure handler is not given, the current figure is used ins
 """

        if figure is None : figure = p.gcf()
        # simple attibute to store the dragged text object
        self.dragged = None

        # Connect events and callbacks
        figure.canvas.mpl_connect("pick_event", self.on_pick_event)
        figure.canvas.mpl_connect("button_release_event", self.on_r

    def on_pick_event(self, event):
        " Store which text object was picked and were the pick even

        if isinstance(event.artist, Text):
            self.dragged = event.artist
            self.pick_pos = (event.mouseevent.xdata, event.mouseeve
        return True

    def on_release_event(self, event):
        " Update text position and redraw"

        if self.dragged is not None :
            old_pos = self.dragged.get_position()
            new_pos = (old_pos[0] + event.xdata - self.pick_pos[0],
                       old_pos[1] + event.ydata - self.pick_pos[1])
            self.dragged.set_position(new_pos)
            self.dragged = None
            p.draw()
        return True
```

A small use case

```
#! python numbers=disable

# Usage example
from numpy import *

# Create arbitrary points and labels
x, y = random.normal(5, 2, size=(2, 9))
labels = [ "Point %d" % i for i in xrange(x.size)]

# trace a scatter plot
p.scatter(x, y)
p.grid()

# add labels and set their picker attribute to True
for a,b,l in zip(x,y, labels):
    p.text(a, b, l, picker=True)

# Create the event hendler
dragh = DragHandler()

p.show()
```

The Text objects can now be moved with the mouse.

Attachments

- `Text_DragnDrop.py`
- `Text_DragnDrop_v0.1.py`
- `Text_DragnDrop_v2.py`

Matplotlib: treemap

Treemaps are a nice way of showing tree information not based on a connected node approach.

See http://www.cs.umd.edu/hcil/treemap/

[](files/attachments/Matplotlib_TreeMap/TreeMap.png

```
"""
Treemap builder using pylab.

Uses algorithm straight from http://hcil.cs.umd.edu/trs/91-03/91-03

James Casbon 29/7/2006
"""

import pylab
from matplotlib.patches import Rectangle

class Treemap:
    def __init__(self, tree, iter_method, size_method, color_method
        """create a tree map from tree, using itermethod(node) to w
 size_method(node) to get object size and color_method(node) to get
 color"""

        self.ax = pylab.subplot(111,aspect='equal')
        pylab.subplots_adjust(left=0, right=1, top=1, bottom=0)
        self.ax.set_xticks([])
        self.ax.set_yticks([])

        self.size_method = size_method
        self.iter_method = iter_method
        self.color_method = color_method
        self.addnode(tree)

    def addnode(self, node, lower=[0,0], upper=[1,1], axis=0):
        axis = axis % 2
        self.draw_rectangle(lower, upper, node)
        width = upper[axis] - lower[axis]
        try:
            for child in self.iter_method(node):
                upper[axis] = lower[axis] + (width * float(size(chi
                self.addnode(child, list(lower), list(upper), axis
                lower[axis] = upper[axis]

        except TypeError:
            pass
```

```
    def draw_rectangle(self, lower, upper, node):
        print lower, upper
        r = Rectangle( lower, upper[0]-lower[0], upper[1] - lower[1
                     edgecolor='k',
                     facecolor= self.color_method(node))
        self.ax.add_patch(r)

if __name__ == '__main__':
    # example using nested lists, iter to walk and random colors

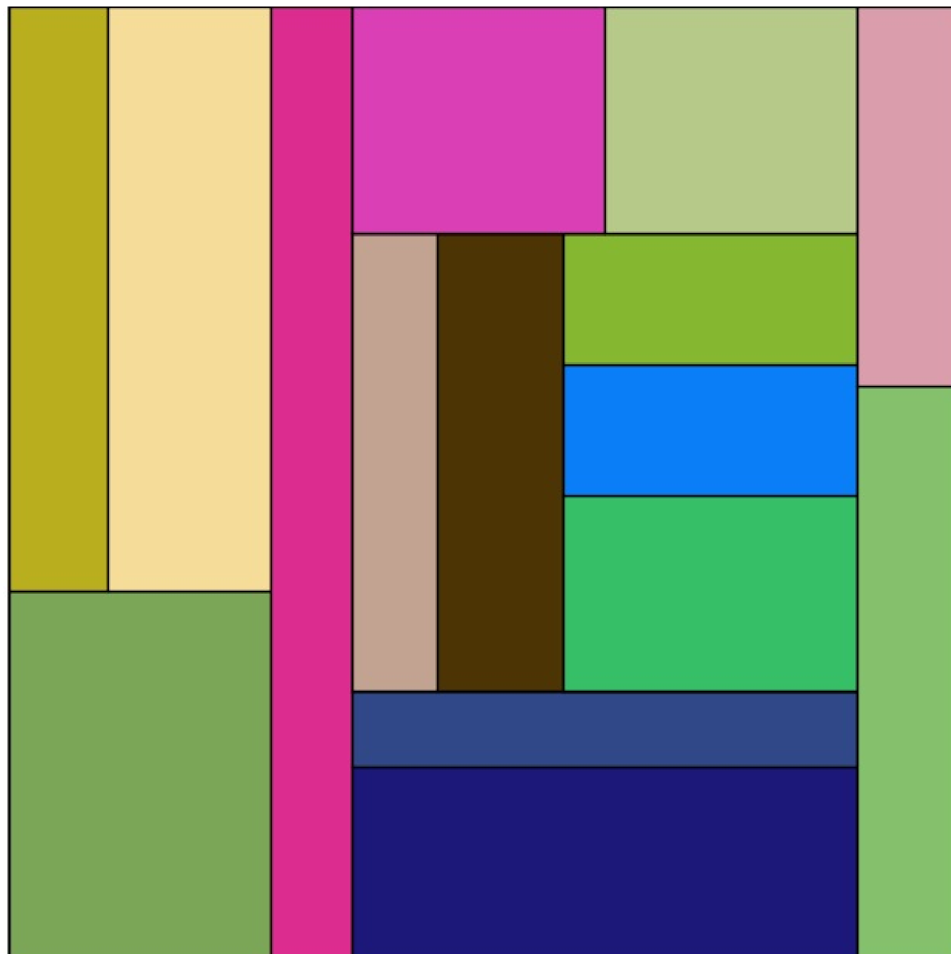
    size_cache = {}
    def size(thing):
        if isinstance(thing, int):
            return thing
        if thing in size_cache:
            return size_cache[thing]
        else:
            size_cache[thing] = reduce(int.__add__, [size(x) for x
            return size_cache[thing]
    import random
    def random_color(thing):
        return (random.random(),random.random(),random.random())

    tree= ((5,(3,5)), 4, (5,2,(2,3,(3,2,2)),(3,3)), (3,2) )

    Treemap(tree, iter, size, random_color)
    pylab.show()
```

Attachments

- `TreeMap.png`



Mayavi: Install python stuff from source

Following http://www.enthought.com/enthought/wiki/GrabbingAndBuilding, you have to build/install VTK 5.0 and a few python extensions from sources.

All needed installation information for a given python module or VTK can be reached on its webpage.

For the impatient, these informations are resumed here.

Note about configure script: If you don't specify the destination where the packages will be installed, they will be installed by defaut in /usr/local.

We make the choice here to install them in a personnal directory, say ~/Mayavi2. So we set the environment variable DESTDIR to ~/Mayavi2, and will refer it later as DESTDIR:

Under sh shell-like, type:

```
export DESTDIR=~/Mayavi2
```

Under csh shell-like, type:

```
setenv DESTDIR ~/Mayavi2
```

Is is also supposed that you download and uncompress all tarball sources in a specific directory, named src/, for example.

Installing python2.3/python2.4

Download Python-2.3.5.tar.bz2 at http://www.python.org/download/releases/2.3.5 or Python-2.4.3.tar.bz2 at http://www.python.org/download/releases/2.4.3 and untar it in src/:

```
cd src && tar xvfj Python-2.4.3.tar.bz2
```

Then run:

```
cd Python-2.4.3/ && ./configure --enable-shared --enable-unicode=uc
```

Then you can make & make install:

```
make && make install
```

Installing VTK 5.0

Download vtk-5.0.0.tar.gz and vtkdata-5.0.0.tar.gz at http://public.kitware.com/VTK/get-software.php and untar them in src/:

```
cd src/ && tar xvfz vtk-5.0.0.tar.gz && tar xvfz vtkdata-5.0.0.tar
```

Note: cmake package must be installed before proceed.

Run:

```
cd VTK && ccmake .
```

to create the required Makefile.

Press on “c” to configure.

Then press “enter” on the selected item to toggle flag.

You should specify some information, notably about some libraries location (tcl/tk libs + dev packages and python2.3/python2.4 you have just installed) if ccmake does not find them, and the destination (set it to DESTDIR).

Don’t forget to set flag “VTK_WRAP_PYTHON” to on (and “VTK_WRAP_TCL” if you want to use Tcl/Tk):

```
BUILD_EXAMPLES                   ON
BUILD_SHARED_LIBS                ON
CMAKE_BACKWARDS_COMPATIBILITY    2.0
CMAKE_BUILD_TYPE
CMAKE_INSTALL_PREFIX             DESTDIR
VTK_DATA_ROOT                    DESTDIR/VTKData
VTK_USE_PARALLEL                 OFF
VTK_USE_RENDERING                ON
VTK_WRAP_JAVA                    OFF
VTK_WRAP_PYTHON                  ON
VTK_WRAP_TCL                     ON
```

Press “c” to continue configuration:

```
PYTHON_INCLUDE_PATH              *DESTDIR/include/python2.4
PYTHON_LIBRARY                   *DESTDIR/lib/libpython2.4.so
TCL_INCLUDE_PATH                 */usr/include/tcl8.4
TCL_LIBRARY                      */usr/lib/libtcl8.4.so
TK_INCLUDE_PATH                  */usr/include/tcl8.4
TK_LIBRARY                       */usr/lib/libtk8.4.so
VTK_USE_RPATH                    *OFF
BUILD_EXAMPLES                    ON
BUILD_SHARED_LIBS                 ON
CMAKE_BACKWARDS_COMPATIBILITY     2.0
CMAKE_BUILD_TYPE
CMAKE_INSTALL_PREFIX              DESTDIR
VTK_DATA_ROOT                     DESTDIR/VTKData
VTK_USE_PARALLEL                  OFF
VTK_USE_RENDERING                 ON
VTK_WRAP_JAVA                     OFF
VTK_WRAP_PYTHON                   ON
VTK_WRAP_TCL                      ON
```

Note: you can press “t” to get more configuration options.

Press “c” and then “g” to exit configuration, then type:

```
make && make install
```

Installing wx-Python2.6

Download wxPython-src-2.6.3.2.tar.gz at https://sourceforge.net/project/showfiles.php?group_id=10718 and untar it in src/:

```
cd src/ && tar xvfz wxPython-src-2.6.3.2.tar.gz
```

Note: You should have GTK 2 installed i.e. you should have libgtk-2.6. `and` *libgtk2.6.*-dev packages installed.

Then run:

```
cd wxPython-src-2.6.3.2/ && ./configure --enable-unicode --with-ope
```

Then you can do:

```
make; make -C contrib/src/animate; make -C contrib/src/gizmos; make
```

or follow instructions on wx-Python2.6 webpage, creating a little script which runs automatically the commands above.

Then install all:

```
make install; make -C contrib/src/animate install ; make -C contrib
```

To build python modules:

```
cd wxPython
```

and run:

```
./setup.py build_ext --inplace --debug UNICODE=1
```

and install them:

```
./setup.py install UNICODE=1 --prefix=$DESTDIR
```

Installing scipy 0.5 & numpy 1.0

Download scipy-0.5.1.tar.gz at http://www.scipy.org/Download

Before installing scipy, you have to download and install:

```
* numpy-1.0.tar.gz (http://sourceforge.net/project/showfiles.php?gr
```

```
* Atlas libraries (you could install it with your packages manager,
```

No special option are required to install these python extensions.

To install these packages in our $DESTDIR, simply change directory and type:

```
./setup.py install --prefix=$DESTDIR
```

That's all, folks !

Before installing !MayaVi2, you have to set some environment variables, to tell !MayaVi2 where python extensions can be found.

Under sh shell-like, type:

```
export PYTHONPATH=$DESTDIR:$PYTHONPATH
export LD_LIBRARY_PATH=$DESTDIR:$LD_LIBRARY_PATH
```

Under csh shell-like, type:

```
setenv PYTHONPATH ${DESTDIR}:${PYTHONPATH}
setenv LD_LIBRARY_PATH ${DESTDIR}:${LD_LIBRARY_PATH}
```

Mayavi: examples

||<#80FF80> This page presents scripting Mayavi2 using the advanced, object-oriented API. Mayavi2 has recently acquired an easy-to-use, thought maybe not as powerful, scripting module: mlab. You are invited to refer to the section of Mayavi2 user guide. ||

Introduction

Here, you will be presented some examples of rendering scenes you can get with !MayaVi2. You are advised to read [:Cookbook/MayaVi/ScriptingMayavi2] to understand what you see, although most examples given here are self-explanatory.

||<#FF8080> Please note that these examples are not up to date. The example gallery for the latest version of Mayavi can be found at http://enthought.github.com/mayavi/mayavi/auto/examples.html. ||

Example using IsoSurface Module (contour.py)

```
#!/usr/bin/env mayavi2

"""This script demonstrates how one can script MayaVi and use its
contour related modules.  Notice the magic line at the top.
"""
# Author: Prabhu Ramachandran <prabhu_r@users.sf.net>
# Copyright (c) 2005-2007, Enthought, Inc.
# License: BSD Style.

# Standard library imports
from os.path import join, dirname

# Enthought library imports
import enthought.mayavi
from enthought.mayavi.sources.vtk_file_reader import VTKFileReader
from enthought.mayavi.filters.threshold import Threshold
from enthought.mayavi.modules.outline import Outline
from enthought.mayavi.modules.grid_plane import GridPlane
from enthought.mayavi.modules.contour_grid_plane import ContourGri
from enthought.mayavi.modules.iso_surface import IsoSurface
from enthought.mayavi.modules.scalar_cut_plane import ScalarCutPla

def contour():
    """The script itself.  We needn't have defined a function but
 having a function makes this more reusable.
```

```
  """
    # 'mayavi' is always defined on the interpreter.
    # Create a new scene.
    mayavi.new_scene()

    # Read a VTK (old style) data file.
    r = VTKFileReader()
    r.initialize(join(dirname(enthought.mayavi.__file__),
                      'examples', 'data', 'heart.vtk'))
    mayavi.add_source(r)

    # Create an outline for the data.
    o = Outline()
    mayavi.add_module(o)

    # Create three simple grid plane modules.
    # First normal to 'x' axis.
    gp = GridPlane()
    mayavi.add_module(gp)
    # Second normal to 'y' axis.
    gp = GridPlane()
    mayavi.add_module(gp)
    gp.grid_plane.axis = 'y'
    # Third normal to 'z' axis.
    gp = GridPlane()
    mayavi.add_module(gp)
    gp.grid_plane.axis = 'z'

    # Create one ContourGridPlane normal to the 'x' axis.
    cgp = ContourGridPlane()
    mayavi.add_module(cgp)
    # Set the position to the middle of the data.
    cgp.grid_plane.position = 15

    # Another with filled contours normal to 'y' axis.
    cgp = ContourGridPlane()
    mayavi.add_module(cgp)
    # Set the axis and position to the middle of the data.
    cgp.grid_plane.axis = 'y'
    cgp.grid_plane.position = 15
    cgp.contour.filled_contours = True

    # An isosurface module.
    iso = IsoSurface(compute_normals=True)
    mayavi.add_module(iso)
    iso.contour.contours = [220.0]

    # An interactive scalar cut plane.
    cp = ScalarCutPlane()
    mayavi.add_module(cp)
    cp.implicit_plane.normal = 0,0,1

if __name__ == '__main__':
```

```
    contour()
```

[](files/attachments/MayaVi_examples/contour.png

Example using Glyph Module (glyph.py)

```
#!/usr/bin/env mayavi2

"""This script demonstrates the use of a VectorCutPlane, splitting
pipeline using a MaskPoints filter and then viewing the filtered da
with the Glyph module.
"""
# Author: Prabhu Ramachandran <prabhu_r@users.sf.net>
# Copyright (c) 2005-2007, Enthought, Inc.
# License: BSD Style.

# Standard library imports
from os.path import join, dirname

# Enthought library imports
import enthought.mayavi
from enthought.mayavi.sources.vtk_xml_file_reader import VTKXMLFile
from enthought.mayavi.modules.outline import Outline
from enthought.mayavi.modules.glyph import Glyph
from enthought.mayavi.modules.vector_cut_plane import VectorCutPlan
from enthought.mayavi.modules.vectors import Vectors
from enthought.mayavi.filters.mask_points import MaskPoints

def glyph():
    """The script itself.  We needn't have defined a function but
 having a function makes this more reusable.
 """
    # 'mayavi' is always defined on the interpreter.
    # Create a new VTK scene.
    mayavi.new_scene()

    # Read a VTK (old style) data file.
    r = VTKXMLFileReader()
    r.initialize(join(dirname(enthought.mayavi.__file__),
                      'examples', 'data', 'fire_ug.vtu'))
    mayavi.add_source(r)

    # Create an outline and a vector cut plane.
    mayavi.add_module(Outline())

    v = VectorCutPlane()
    mayavi.add_module(v)
    v.glyph.color_mode = 'color_by_scalar'
```

```
    # Now mask the points and show glyphs (we could also use
    # Vectors but glyphs are a bit more generic)
    m = MaskPoints()
    m.filter.set(on_ratio=10, random_mode=True)
    mayavi.add_filter(m)

    g = Glyph()
    mayavi.add_module(g)
    # Note that this adds the module to the filtered output.
    g.glyph.scale_mode = 'scale_by_vector'
    # Use arrows to view the scalars.
    g.glyph.glyph_source = g.glyph.glyph_list[1]

if __name__ == '__main__':
    glyph()
```

[](files/attachments/MayaVi_examples/glyph.png

Example without Mayavi2 UI (nongui.py)

```
#!/usr/bin/env python

"""This script demonstrates how one can use the MayaVi framework
without displaying MayaVi's UI.  Note: look at the end of this file
see how the non gui plugin is chosen instead of the default gui
mayavi plugin.

"""
# Author: Prabhu Ramachandran <prabhu_r@users.sf.net>
# Copyright (c) 2005, Enthought, Inc.
# License: BSD Style.

# On systems with multiple wx installations installed, pick one tha
# with the libraries Mayavi depends on.
try:
    import wxversion
    wxversion.ensureMinimal('2.6')
except ImportError:
    pass

# Standard library imports
import sys
from os.path import join, dirname

# Enthought library imports
from enthought.mayavi.app import Mayavi, NONGUI_PLUGIN_DEFINITIONS

class MyApp(Mayavi):
    def run(self):
```

```
        """This is executed once the application GUI has started.
*Make sure all other MayaVi specific imports are made here!*
"""

        # Various imports to do different things.
        from enthought.mayavi.sources.vtk_file_reader import VTKFi
        from enthought.mayavi.modules.outline import Outline
        from enthought.mayavi.modules.axes import Axes
        from enthought.mayavi.modules.grid_plane import GridPlane
        from enthought.mayavi.modules.image_plane_widget import Ima
        from enthought.mayavi.modules.text import Text
        from enthought.mayavi.modules.contour_grid_plane import Cor
        from enthought.mayavi.modules.iso_surface import IsoSurface

        script = self.script

        # Create a new scene.
        script.new_scene()

        # Read a VTK (old style) data file.
        r = VTKFileReader()
        r.initialize('data/heart.vtk')
        r.initialize(join(dirname(__file__), 'data', 'heart.vtk'))
        script.add_source(r)

        # Put up some text.
        t = Text(text='MayaVi rules!', x_position=0.2, y_position=0
        t.property.color = 1, 1, 0  # Bright yellow, yeah!
        script.add_module(t)

        # Create an outline for the data.
        o = Outline()
        script.add_module(o)

        # Create an axes for the data.
        a = Axes()
        script.add_module(a)

        # Create three simple grid plane modules.
        # First normal to 'x' axis.
        gp = GridPlane()
        script.add_module(gp)
        # Second normal to 'y' axis.
        gp = GridPlane()
        gp.grid_plane.axis = 'y'
        script.add_module(gp)
        # Third normal to 'z' axis.
        gp = GridPlane()
        script.add_module(gp)
        gp.grid_plane.axis = 'z'

        # Create one ImagePlaneWidget.
        ipw = ImagePlaneWidget()
```

```
        script.add_module(ipw)
        # Set the position to the middle of the data.
        ipw.ipw.slice_position = 16

        # Create one ContourGridPlane normal to the 'x' axis.
        cgp = ContourGridPlane()
        script.add_module(cgp)
        # Set the position to the middle of the data.
        cgp.grid_plane.axis = 'y'
        cgp.grid_plane.position = 15

        # An isosurface module.
        iso = IsoSurface(compute_normals=True)
        script.add_module(iso)
        iso.contour.contours = [200.0]

        # Set the view.
        s = script.engine.current_scene
        cam = s.scene.camera
        cam.azimuth(45)
        cam.elevation(15)
        s.render()

if __name__ == '__main__':
    m = MyApp()
    # Note how we change the plugins that are loaded only here.
    m.main(plugin_defs=NONGUI_PLUGIN_DEFINITIONS)
```

[](files/attachments/MayaVi_examples/nongui.png

Example with a 3D array as numerical source (numeric_source.py)

```
#!/usr/bin/env mayavi2

"""This script demonstrates how to create a numpy array data and
visualize it as image data using a few modules.

"""
# Author: Prabhu Ramachandran <prabhu_r@users.sf.net>
# Copyright (c) 2005-2007, Enthought, Inc.
# License: BSD Style.

# Standard library imports
import enthought.util.scipyx as scipy

# Enthought library imports
from enthought.mayavi.sources.array_source import ArraySource
```

```
from enthought.mayavi.modules.outline import Outline
from enthought.mayavi.modules.image_plane_widget import ImagePlaneW

def make_data(dims=(128, 128, 128)):
    """Creates some simple array data of the given dimensions to te
 with."""
    np = dims[0]*dims[1]*dims[2]

    # Create some scalars to render.
    x, y, z = scipy.ogrid[-5:5:dims[0]*1j,-5:5:dims[1]*1j,-5:5:dims
    x = x.astype('f')
    y = y.astype('f')
    z = z.astype('f')

    scalars = (scipy.sin(x*y*z)/(x*y*z))
    return scipy.transpose(scalars).copy() # This makes the data co

def view_numpy():
    """Example showing how to view a 3D numpy array in mayavi2.
 """
    # 'mayavi' is always defined on the interpreter.
    mayavi.new_scene()
    # Make the data and add it to the pipeline.
    data = make_data()
    src = ArraySource(transpose_input_array=False)
    src.scalar_data = data
    mayavi.add_source(src)
    # Visualize the data.
    o = Outline()
    mayavi.add_module(o)
    ipw = ImagePlaneWidget()
    mayavi.add_module(ipw)
    ipw.module_manager.scalar_lut_manager.show_scalar_bar = True

    ipw_y = ImagePlaneWidget()
    mayavi.add_module(ipw_y)
    ipw_y.ipw.plane_orientation = 'y_axes'

if __name__ == '__main__':
    view_numpy()
```

[](files/attachments/MayaVi_examples/numeric_source.png

Example using Streamline Module (streamline.py)

```
#!/usr/bin/env mayavi2
"""This script demonstrates how one can script MayaVi to display
streamlines and an iso surface.
"""
```

```
# Author: Prabhu Ramachandran <prabhu_r@users.sf.net>
# Copyright (c) 2005-2007, Enthought, Inc.
# License: BSD Style.

# Standard library imports
from os.path import join, dirname

# Enthought library imports
from enthought.mayavi.sources.vtk_xml_file_reader import VTKXMLFile
from enthought.mayavi.modules.outline import Outline
from enthought.mayavi.modules.streamline import Streamline
from enthought.mayavi.modules.iso_surface import IsoSurface

def setup_data(fname):
    """Given a VTK XML file name `fname`, this creates a mayavi2
 reader for it and adds it to the pipeline.  It returns the reader
 created.
 """
    mayavi.new_scene()
    r = VTKXMLFileReader()
    r.initialize(fname)
    mayavi.add_source(r)
    return r

def streamline():
    """Sets up the mayavi pipeline for the visualization.
 """
    # Create an outline for the data.
    o = Outline()
    mayavi.add_module(o)

    s = Streamline(streamline_type='tube')
    mayavi.add_module(s)
    s.stream_tracer.integration_direction = 'both'
    s.seed.widget.center = 3.5, 0.625, 1.25
    s.module_manager.scalar_lut_manager.show_scalar_bar = True

    i = IsoSurface()
    mayavi.add_module(i)
    i.contour.contours[0] = 550
    i.actor.property.opacity = 0.5

if __name__ == '__main__':
    import enthought.mayavi
    fname = join(dirname(enthought.mayavi.__file__),
                 'examples', 'data', 'fire_ug.vtu')
    r = setup_data(fname)
    streamline()
```

[](files/attachments/MayaVi_examples/streamline.png

Example using ImagePlaneWidget Module (test.py)

```
#!/usr/bin/env python

"""This script demonstrates how one can script MayaVi, set its size
create a new VTK scene and create a few simple modules.

"""
# Author: Prabhu Ramachandran <prabhu_r@users.sf.net>
# Copyright (c) 2005, Enthought, Inc.
# License: BSD Style.

# On systems with multiple wx installations installed, pick one tha
# with the libraries Mayavi depends on.
try:
    import wxversion
    wxversion.ensureMinimal('2.6')
except ImportError:
    pass

# Standard library imports
import sys
from os.path import join, dirname

# Enthought library imports
from enthought.mayavi.app import Mayavi

class MyApp(Mayavi):
    def run(self):
        """This is executed once the application GUI has started.
 *Make sure all other MayaVi specific imports are made here!*
 """
        # Various imports to do different things.
        from enthought.mayavi.sources.vtk_file_reader import VTKFil
        from enthought.mayavi.filters.threshold import Threshold
        from enthought.mayavi.modules.outline import Outline
        from enthought.mayavi.modules.axes import Axes
        from enthought.mayavi.modules.grid_plane import GridPlane
        from enthought.mayavi.modules.image_plane_widget import Ima
        from enthought.mayavi.modules.text import Text

        script = self.script
        # Create a new scene.
        script.new_scene()

        # Read a VTK (old style) data file.
        r = VTKFileReader()
        r.initialize(join(dirname(__file__), 'data', 'heart.vtk'))
        script.add_source(r)

        # Put up some text.
```

```
        t = Text(text='MayaVi rules!', x_position=0.2,
                 y_position=0.9, width=0.8)
        t.property.color = 1, 1, 0  # Bright yellow, yeah!
        script.add_module(t)

        # Create an outline for the data.
        o = Outline()
        script.add_module(o)

        # Create an axes for the data.
        a = Axes()
        script.add_module(a)

        # Create an orientation axes for the scene.  This only work
        # VTK-4.5 and above which is why we have the try block.
        try:
            from enthought.mayavi.modules.orientation_axes import O
        except ImportError:
            pass
        else:
            a = OrientationAxes()
            a.marker.set_viewport(0.0, 0.8, 0.2, 1.0)
            script.add_module(a)

        # Create three simple grid plane modules.
        # First normal to 'x' axis.
        gp = GridPlane()
        script.add_module(gp)
        # Second normal to 'y' axis.
        gp = GridPlane()
        gp.grid_plane.axis = 'y'
        script.add_module(gp)
        # Third normal to 'z' axis.
        gp = GridPlane()
        script.add_module(gp)
        gp.grid_plane.axis = 'z'

        # Create one ImagePlaneWidget.
        ipw = ImagePlaneWidget()
        script.add_module(ipw)
        # Set the position to the middle of the data.
        ipw.ipw.slice_position = 16

if __name__ == '__main__':
    a = MyApp()
    a.main()
```

[](files/attachments/MayaVi_examples/test.png

Example using mlab (surf_regular_mlab.py)

See also [:Cookbook/MayaVi/Surf] for another way of doing this.

```
#!/usr/bin/env mayavi2
"""Shows how to view data created by `enthought.tvtk.tools.mlab` wi
mayavi2.
"""

# Author: Prabhu Ramachandran <prabhu@aero.iitb.ac.in>
# Copyright (c) 2006-2007, Enthought Inc.
# License: BSD Style.

import numpy

from enthought.tvtk.tools import mlab
from enthought.mayavi.sources.vtk_data_source import VTKDataSource
from enthought.mayavi.filters.warp_scalar import WarpScalar
from enthought.mayavi.modules.outline import Outline
from enthought.mayavi.modules.surface import Surface

def f(x, y):
    """Some test function.
 """
    return numpy.sin(x*y)/(x*y)

def make_data():
    """Make some test numpy data and create a TVTK data object fror
 that we will visualize.
 """
    x = numpy.arange(-7., 7.05, 0.1)
    y = numpy.arange(-5., 5.05, 0.05)
    s = mlab.SurfRegular(x, y, f)
    return s.data

def add_data(tvtk_data):
    """Add a TVTK data object `tvtk_data` to the mayavi pipleine.
 """
    d = VTKDataSource()
    d.data = tvtk_data
    mayavi.add_source(d)

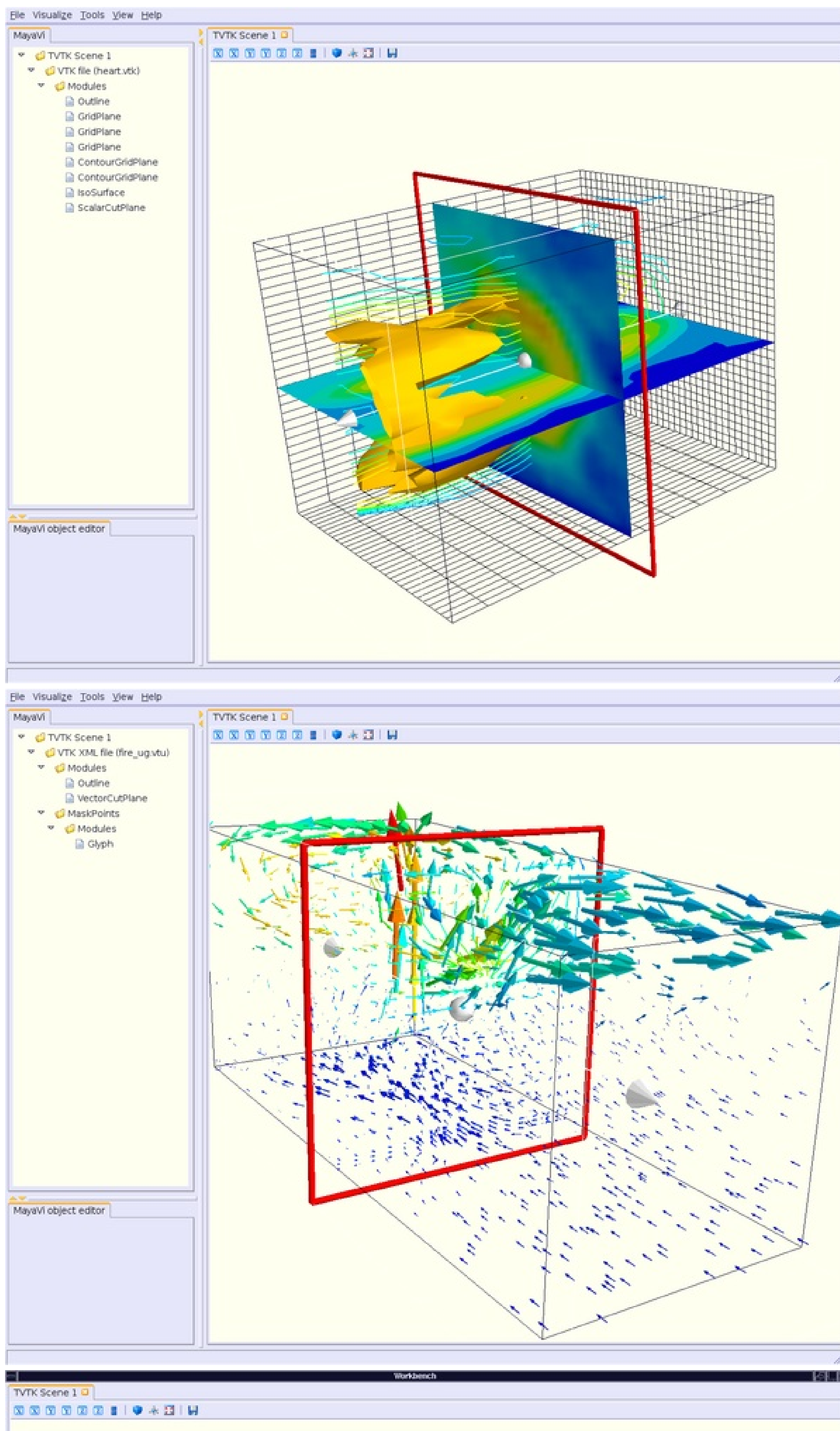
def surf_regular():
    """Now visualize the data as done in mlab.
 """
    w = WarpScalar()
    mayavi.add_filter(w)
    o = Outline()
    s = Surface()
    mayavi.add_module(o)
    mayavi.add_module(s)

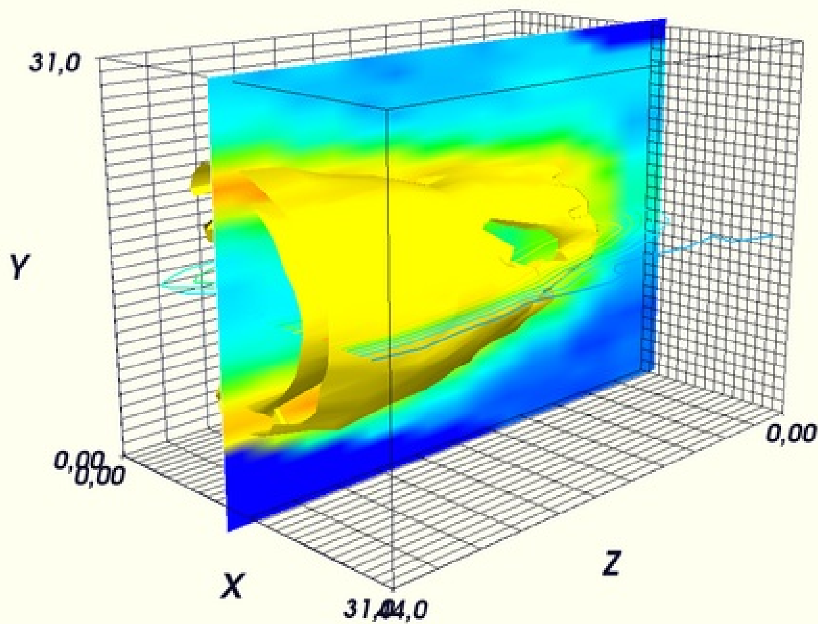
if __name__ == '__main__':
    mayavi.new_scene()
```

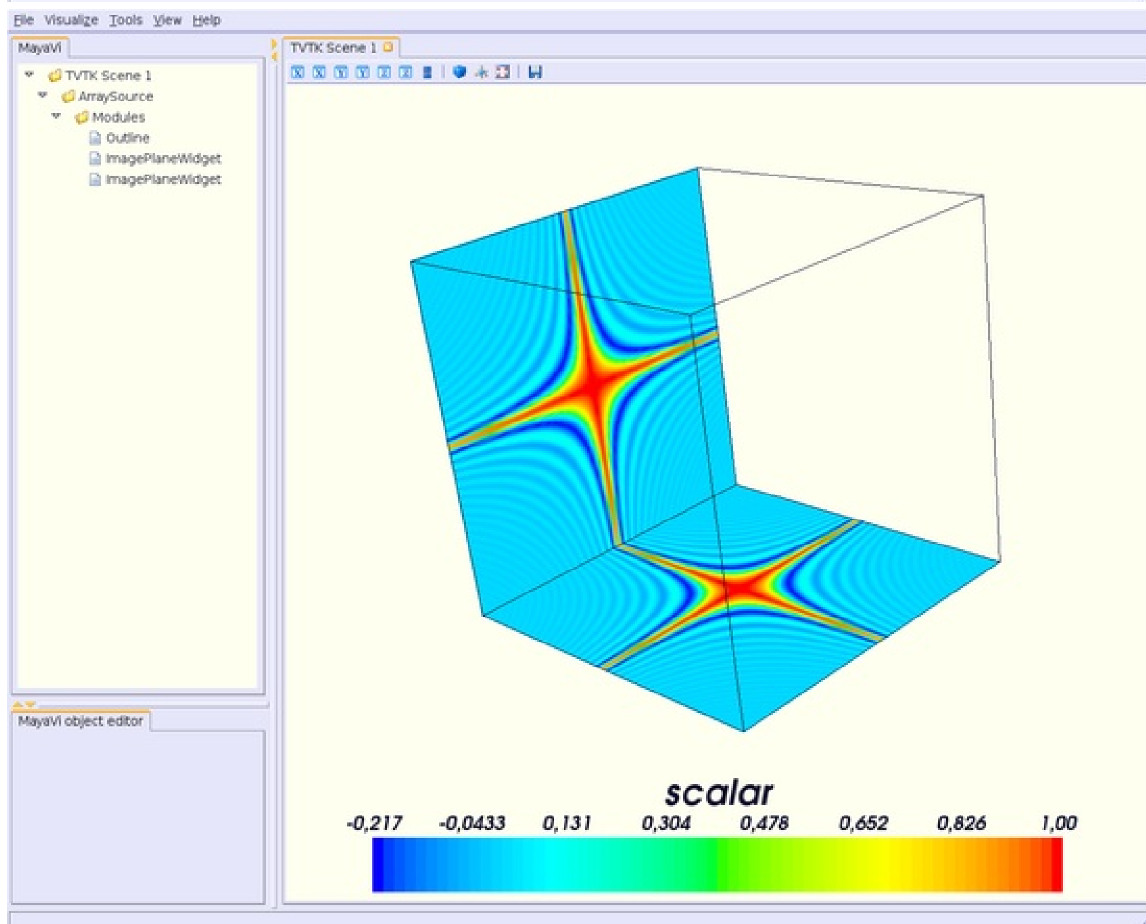
```
    d = make_data()
    add_data(d)
    surf_regular()
```

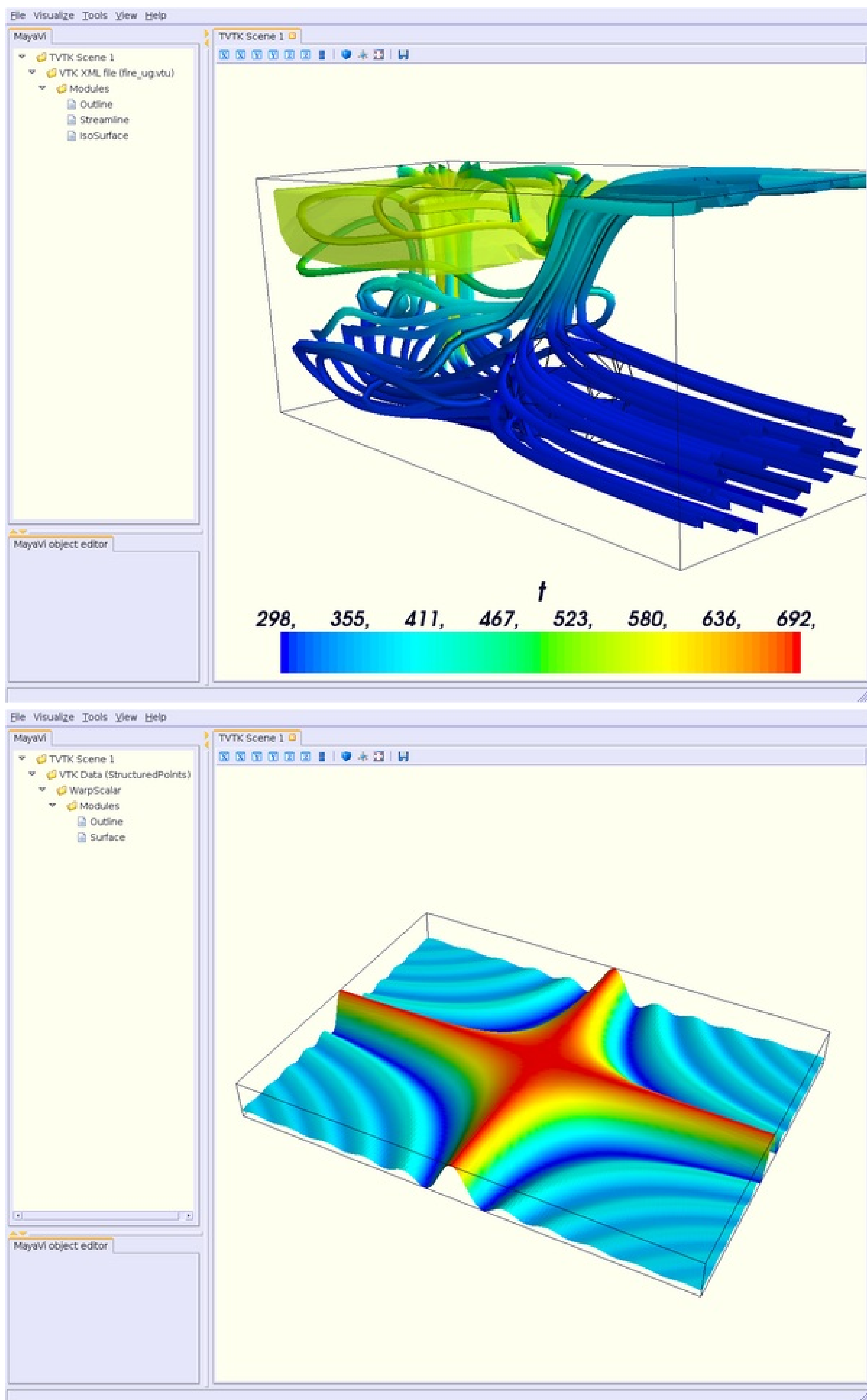
Attachments

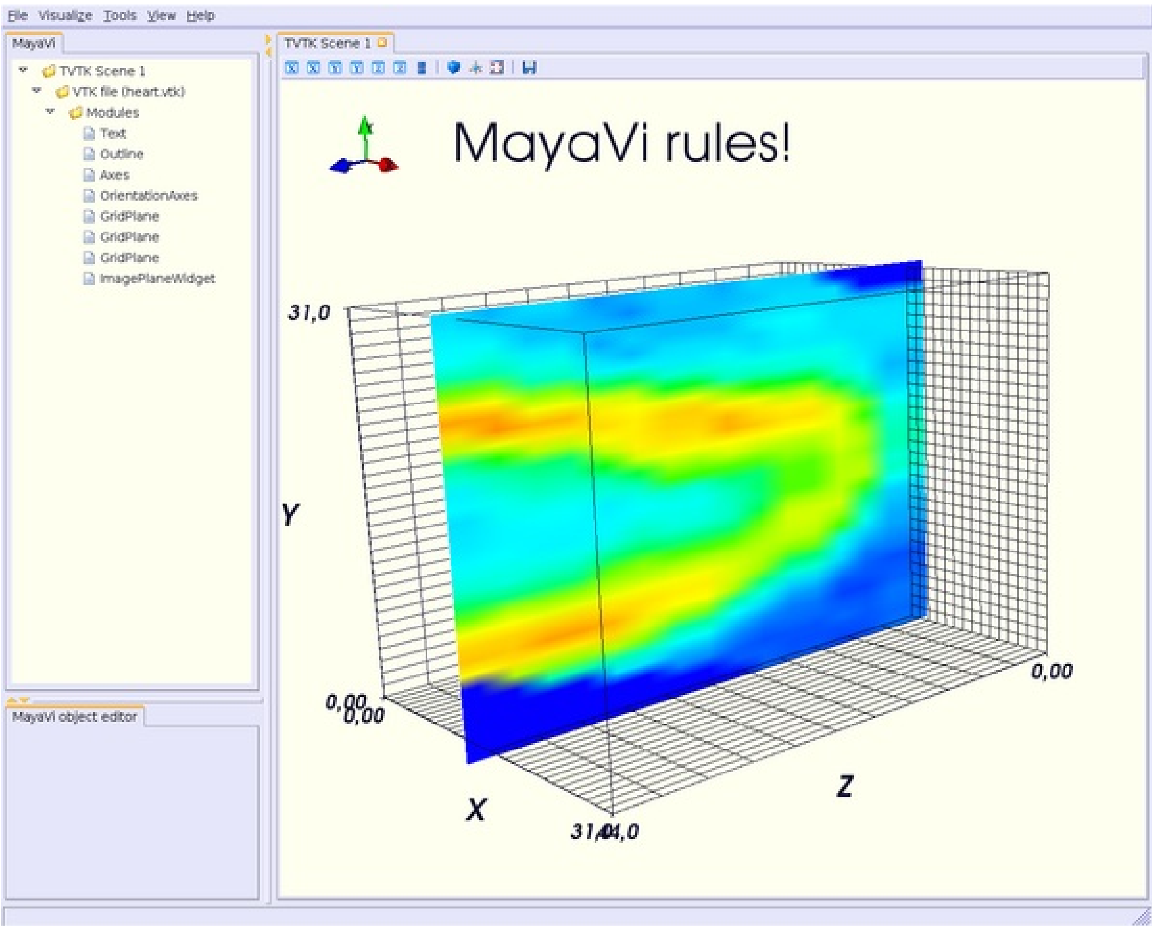
- `contour.png`
- `glyph.png`
- `nongui.png`
- `numeric_source.png`
- `streamline.png`
- `surf_regular_mlab.png`
- `test.png`


File Visualize Tools View Help
MayaVi
TVTK Scene 1
VTK file (heart.vtk)
Modules
Outline
GridPlane
GridPlane
GridPlane
ContourGridPlane
ContourGridPlane
IsoSurface
ScalarCutPlane
MayaVi object editor
File Visualize Tools View Help
MayaVi
TVTK Scene 1
VTK XML file (fire_ug.vtu)
Modules
Outline
VectorCutPlane
MaskPoints
Modules
Glyph
MayaVi object editor
TVTK Scene 1


MayaVi rules!
31,0
Y
0,00
X
Z
0,00


File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Modules
Outline
ImagePlaneWidget
ImagePlaneWidget
MayaVi object editor
scalar
-0,217
-0,0433
0,131
0,304
0,478
0,652
0,826
1,00


File Visualize Tools View Help
MayaVi
TVTK Scene 1
VTK XML file (fire_ug.vtu)
Modules
Outline
Streamline
IsoSurface
MayaVi object editor
t
298, 355, 411, 467, 523, 580, 636, 692,
File Visualize Tools View Help
MayaVi
TVTK Scene 1
VTK Data (StructuredPoints)
WarpScalar
Modules
Outline
Surface
MayaVi object editor


File Visualize Tools View Help
MayaVi
TVTK Scene 1
VTK file (heart.vtk)
Modules
Text
Outline
Axes
OrientationAxes
GridPlane
GridPlane
GridPlane
ImagePlaneWidget
MayaVi object editor
MayaVi rules!
31,0
Y
0,00
0,00
X
31,0
Z
0,00

Reading custom text files with Pyparsing

Introduction

In this cookbook, we will focus on using pyparsing and numpy to read a structured text file like this one, `data.txt` :

```
# This is is an example file structured in section
# with comments begining with '#'

[ INFOS ]
Debug           = False
Shape  (mm^-1) = 2.3
Length (mm)    = 25361.15
Path 1         = C:\\This\is\a\long\path\with some space in it\data
description    = raw values can have multiple lines, but additional
                 with a whitespace which is automatically skipped
Parent         = None

[ EMPTY SECTION ]
# empty section should not be taken into account

[ TABLE IN ROWS ]
Temp    (C)             100           200        300        450.0
E XX    (GPa)           159.4       16.9E+0       51.8      .15E02
Words               'hundred'   'two hundreds'  'a lot'     'four'

[ TABLE IN COLUMNS ]
STATION         PRECIPITATION   T_MAX_ABS  T_MIN_ABS
(/)                     (mm)    (C)        (C)        # Columns must
Ajaccio                 64.8    18.8E+0    -2.6
Auxerre                 49.6    16.9E+0    Nan        # Here is a Na
Bastia                  114.2   20.8E+0    -0.9

[ MATRIX ]
True    2       3
4\.      5\.      6.
7\.      nan     8
```

and we will create a reusable parser class to automatically:

```
 * detect section blocs, among four possible kinds :`` * a set of var
```

Here is a session example with this parser, `ConfigNumParser` :)#

```
>>> from ConfigNumParser import *
>>> data = parseConfigFile('data.txt')
>>> pprint(data.asList())
[['infos',
  ['debug', False],
  ['shape', 2.2999999999999998],
  ['length', 25361.150000000001],
  ['path_1', 'C:\\\\This\\is\\a\\long\\path\\with some space in it\\
  ['description',
  'raw values can have multiple lines, but additional lines must sta
  ['parent', None],
  ['names_', ['debug', 'shape', 'length', 'path_1', 'description', 
  ['unit_', {'length': 'mm', 'shape': 'mm^-1'}]],
  ['table_in_rows',
  ['temp', array([ 100.,  200.,  300.,  450.,  600.])],
  ['e_xx', array([ 159.4,   16.9,   51.8,   15\. ,    4\. ])],
  ['words', array(['hundred', 'two hundreds', 'a lot', 'four', 'five
  ['names_', ['temp', 'e_xx', 'words']],
  ['unit_', {'e_xx': 'GPa', 'temp': 'C'}]],
  ['table_in_columns',
  ['station', array(['Ajaccio', 'Auxerre', 'Bastia'], dtype='|S7')],
  ['precipitation', array([  64.8,   49.6,  114.2])],
  ['t_max_abs', array([ 18.8,  16.9,  20.8])],
  ['t_min_abs', array([-2.6,  NaN, -0.9])],
  ['names_', ['station', 'precipitation', 't_max_abs', 't_min_abs']
  ['unit_',  {'precipitation': 'mm', 't_max_abs': 'C', 't_min_abs': 
  ['matrix',
  array([[  1.,   2.,   3.],
  [  4.,   5.,   6.],
  [  7.,  NaN,   8.]])]]

>>> data.matrix
array([[  1.,   2.,   3.],
 [  4.,   5.,   6.],
 [  7.,  NaN,   8.]])

>>> data.table_in_columns.t_max_abs
array([ 18.8,  16.9,  20.8])

>>> data.infos.length, data.infos.unit_['length']
(25361.15, 'mm')
```

This parser add two specials fields in all sections but matrix ones :

`* `` `*names*` _` : a list containing the names of all variables found in`
* `_``unit_``_` : a dict containing the unit corresponding to each variable name, if there is any`

Defining a parser for parameter declarations

pyparsing is an efficient tool to deal with formatted text, and let you process in two steps:

```
1. Define rules to identify strings representing sections, variable
```

```
1. Define actions to be executed on theses fields, to convert them i
```

In the file example above, there are four kinds of data: parameter definitions, table in rows, table in columns and matrix.

So, we will define a parser for each one and combine them to define the final parser.

First steps with pyparsing

This section will describe step by step how to build the function `paramParser` defined in `ConfigNumParser`, used to parse)# the bloc [INFOS] in the example above.

A parameter declaration has the form:

```
key ( unit ) = value with:`` * key : a set of alphanumeric ch
```

This can be translated almost literally with pyparsing syntax (see how to use pyparsing for more information):

```
from    pyparsing   import *
# parameter definition
keyName       = Word(alphanums + '_')
unitDef       = '(' + Word(alphanums + '^*/-._') + ')'
paramValueDef = SkipTo('#'|lineEnd)

paramDef = keyName + Optional(unitDef) + "=" + empty + paramValueDe
```

It is easy to test what will be found with this pattern in the data file:

```
# print all params found
>>> for param in paramDef.searchString(file('data.txt').read()):
...     print param.dump()
...     print '...'
['Context', '=', 'full']
...
['Temp_ref', '(', 'K', ')', '=', '298.15']
...
...
```

We can improved it in a few ways:

```
 * suppress meaningless fields '(', '=', ')' from the output, with tl
```
Suppress `element, ``* give a name to the different fields, with the`
setResultsName `method, or simply just by calling an element with the`

```
# parameter definition
keyName       = Word(alphanums + '_')
unitDef       = Suppress('(') + Word(alphanums + '^*/-._') + Suppre
paramValueDef = SkipTo('#'|lineEnd)

paramDef = keyName('name') + Optional(unitDef)('unit') + Suppress('
```

The test will now give name to results and gives a nicer output:

```
['Context', 'full']
- name: Context
- value: full
...
['Temp_ref', 'K', '298.15']
- name: Temp_ref
- unit: ['K']
- value: 298.15
...
...
```

Converting data into Python objects

We will detail further what kind of values are expected to let pyparsing handle the conversion.

They can be divided in two parts :

```
 * Python objects like numbers, True, False, None, NaN or any string
```

Let’s begin with numbers. We can use the `Regex` element to rapidly detect strings representing numbers:

```
from re         import VERBOSE
number = Regex(r"""
 [+-]?                          # optional sign
 (
 (?:\d+(?P<float1>\.\d*)?)   # match 2 or 2.02
 |                             # or
 (?P<float2>\.\d+)           # match .02
 )
 (?P<float3>[Ee][+-]?\d+)?      # optional exponent
 """, flags=VERBOSE
        )
```

See Regular expression operations for more information on regular expressions. We could have built a parser with standard pyparsing elements (`Combine` , `Optional` , `oneOf` , etc.) but low-level expressions like floating point numbers are said to do really much better using the `Regex` class. I know it feels like cheating, but in truth, pyparsing uses a number of re's under the covers.

Now we will define a function to convert this string into python float or integer and set a `parseAction` to tell pyparsing to automatically convert a number when it find one:

```
def convertNumber(t):
    """Convert a string matching a number to a python number"""
    if t.float1 or t.float2 or t.float3 : return [float(t[0])]
    else                                : return [int(t[0])  ]

number.setParseAction(convertNumber)
```

The `convertNumber` function is a simple example of `parseAction` :

` * it should accepts a `parseResults` object as input value (some funct`
setParseAction ` documentation). A `parseResults` object can be used as a l`

` * it should return either a `parseResults` object or a list of results`
parseResults ` object.`

Pyparsing comes with a very convenient function to convert fields to a constant object, namely `replaceWith` . This can be used to create a list of element converting strings to python objects:

```
from numpy     import NAN

pyValue_list = [ number
                 Keyword('True').setParseAction(replaceWith(True))
                 Keyword('False').setParseAction(replaceWith(False
                 Keyword('NAN', caseless=True).setParseAction(repla
                 Keyword('None').setParseAction(replaceWith(None))
                 QuotedString('"""', multiline=True)
                 QuotedString("'''", multiline=True)
                 QuotedString('"')
                 QuotedString("'")
               ]

pyValue     = MatchFirst( e.setWhitespaceChars(' \t\r') for e in py
```

Here we used:

` * `Keyword` to detect standard python keyword and replace them on the`
QuotedString` to detect quoted string and automatically unquote them``
MatchFirst` to build a super element, `pyValue` to convert all kind of py

Let's see what we get:

```
>>> test2 = '''
>>>     1   2   3.0  0.3 .3  2e2  -.2e+2 +2.2256E-2
>>>     True False nan NAN None
>>>     "word" "two words"
>>>     """'more words', he said"""
>>> '''
>>> print pyValue.searchString(test2)
[[1], [2], [3.0], [0.29999999999999999], [0.29999999999999999], [20
[True], [False], [nan], [nan], [None], ['word'], ['two words'], ["
```

Some words on whitespace characters

By default, pyparsing considers any characters in ' \t\r\n') as whitespace and meaningless. If you need to detect ends-of-line you need to change this behavior by using `setWhitespaceChars` or `setDefaultWhitespaceChars`.

As we are going to process tables line by line, we need to configure this and this should be set up *at the lowest level*:

```
>>> pyValue2     = MatchFirst(pyValue_list)          # default beha
>>> print OneOrMore(pyValue2).searchString(test2)
[[1, 2, 3.0, 0.29999999999999999, 0.29999999999999999, 200.0, -20.0

>>> # to compare to

>>> for r, s, t in OneOrMore(pyValue).searchString(test2)
[[1, 2, 3.0, 0.29999999999999999, 0.29999999999999999, 200.0, -20.0
[True, False, nan, nan, None],
['word', 'two words'],
["'more words', he said"]]
```

Converting variables names

We must also detail what is an acceptable parameter name.

As the end of the parameter name is delimited by the = character, we could accept to have spaces in it. But as we want the possibility to access to its value via a named attribute, we need to convert it to a standard form, compatible with python's naming conventions. Here we choose to format parameter names to lowercase, with any set of character in ' -/.' replaced with underscores.

Later, we will have to deal with parameter names where spaces can't be allowed. So we will have to define two kind of names:

```
def variableParser(escapedChars, baseChars=alphanums):
    """ Return pattern matching any characters in baseChars separat
  characters defined in escapedChars. Thoses characters are replacec

  The '_' character is therefore automatically in escapedChars.
  """
    escapeDef = Word(escapedChars + '_').setParseAction(replaceWitl
    whitespaceChars = ''.join( x for x in ' \t\r' if not x in escap
    escapeDef = escapeDef.setWhitespaceChars(whitespaceChars)
    return Combine(Word(baseChars) + Optional(OneOrMore(escapeDef -

keyName             = variableParser(' _-./').setParseAction(downca
keyNameWithoutSpace = variableParser('_-./').setParseAction(downcas
```

`downcaseTokens` is a special pyparsing function returning every matching tokens lowercase.

Dealing with raw text

To finish this parser, we now need to add a rule to match raw text following the conditions:

```
* anything after the # character is considered as a comment and ski
```

```
# rawValue can be multiline but theses lines should start with a Wh
rawLine  = CharsNotIn('#\n') + (lineEnd | Suppress('#'+restOfLine)
rawValue = Combine( rawLine + ZeroOrMore(White(' \t').suppress()+ N
rawValue.setParseAction(lambda t: [x.strip() for x in t])
```

We will also refine our definition of units to handle special cases like (-), (/) or (), corresponding to a blank unit.

This leads to:

```
unitDef  = Suppress('(') + (Suppress(oneOf('- /')) | Optional(Word(
valueDef = pyValue | rawValue
paramDef = keyName('name') + Optional(unitDef)('unit') + Suppress('
```

Structuring data

We will try to organize the results in an easy to use data structure.

To do so, we will use the `Dict` element, which allows access by key as a normal dict or by named attributes. This element takes for every tokens found, its first field as the key name and the following ones as values. This is very handy when you can group data with the `Group` element to have only two fields.

As we can have three of them (with units) we will put these units aside:

```
def formatBloc(t):
    """ Format the result to have a list of (key, values) easily us

  Add two fields :
  names_ : the list of column names found
  units_ : a dict in the form {key : unit}
  """
    rows = []

    # store units and names
    units = {}
    names = []

    for row in t :
        rows.append(ParseResults([ row.name, row.value ]))
        names.append(row.name)
        if row.unit : units[row.name] = row.unit[0]

    rows.append( ParseResults([ 'names_', names ]))
    rows.append( ParseResults([ 'unit_',  units]))

    return rows

paramParser = Dict( OneOrMore( Group(paramDef)).setParseAction(forr
```

This `paramParser` element is exactly the parser created by the function `paramParser` defined in `ConfigNumParser`.

Let’s see what we get:

```
>>> paramParser.ignore('#' + restOfLine)
>>> data = paramParser.searchString(file('data.txt').read())[0]
>>> print data.dump()
[...]
- debug: False
- description: raw values can have multiple lines, but additional l
with a whitespace which is automatically skipped
- length: 25361.15
- names_: ['debug', 'shape', 'length', 'path_1', 'description', 'pa
- parent: None
- path_1: 'C:\\This\is\a\long\path\with some space in it\data.txt'
- shape: 2.3
- unit_: {'shape': 'mm^-1', 'length': 'mm'}
>>> data.length, data.unit_['length']
Out[12]: (25361.150000000001, 'mm')
```

Defining a parser for tables

For parsing parameter declarations, we have seen most of the common techniques but one: the use of `Forward` element to define parsing rules on the fly.

Let's see how this can be used to parse a table defined column by column, according to this schema:

```
Name_1       Name_2      ...      Name_n
(unit_1)    (unit_2)     ...     (unit_n)
value_11    value_21     ...     value_n1
  ...          ...       ...       ...
```

and the following rules:

```
* Names can't contains any whitespaces.``* Units are mandatory.``* V
```

Such a parser can be generated with the `tableColParser` function defined in `ConfigNumParser` .

The heart of the problem is to tell pyparsing that each line should have the same number of columns, whereas this number is unknown a priori.

Using the Forward element

We will get round this problem by defining the pattern corresponding to the unit line and its followers right after reading the header line.

Indeed, theses lines can be defined with a `Forward` element and we can attach a `parseAction` to the header line to redefine these elements later, once we know how many columns we have in the headers.

Redefining a `Forward` element is done via the << operator:

```
# We define ends-of-line and what kind of values we expect in table
EOL           = LineEnd().suppress()
tabValueDef  = pyValue | CharsNotIn('[ \t\r\n').setWhitespaceChars(

# We define how to detect the first line, which is a header line
# following lines will be defined later
firstLine    = Group(OneOrMore(keyNameWithoutSpace)+EOL)
unitLine     = Forward()
tabValueLine = Forward()

def defineColNumber(t):
    """ Define unitLine and tabValueLine to match the same number c
 the header line"""
    nbcols = len(t.header)
    unitLine      << Group( unitDef*nbcols + EOL)
    tabValueLine  << Group( tabValueDef*nbcols + EOL)

tableColDef = (   firstLine('header').setParseAction(defineColNumbe
                + unitLine('unit')
                + Group(OneOrMore(tabValueLine))('data')
              )
```

Structuring our data

Now will organize our data the same way we did for parameters, but we will use this time the name of the column as the key and we will transform our data into numpy arrays:

```
def formatBloc(t):
    """ Format the result to have a list of (key, values) easily us
 with Dict and transform data into array

 Add two fields :
 names_ : the list of column names found
 units_ : a dict in the form {key : unit}
 """
    columns = []

    # store names and units names
    names = t.header
    units   = {}

    transposedData = zip(*t.data)
    for header, unit, data in zip(t.header, t.unit, transposedData)
        units[header] = unit
        columns.append(ParseResults([header, array(data)]))

    columns.append(ParseResults(['names_', names]))
    columns.append(ParseResults(['unit_'   , units  ]))

    return columns

tableColParser = Dict(tableColDef.setParseAction(formatBloc))
```

Let’s see what we get:

```
>>> tableColParser.ignore('#' + restOfLine)
>>> data = tableColParser.searchString(file('data3.txt').read())[0]
>>> print data.dump()
[...]
- names_: ['station', 'precipitation', 't_max_abs', 't_min_abs']
- precipitation: [  64.8   49.6  114.2]
- station: ['Ajaccio' 'Auxerre' 'Bastia']
- t_max_abs: [ 18.8  16.9  20.8]
- t_min_abs: [-2.6  NaN -0.9]
- unit_: {'station': '/', 'precipitation': 'mm', 't_min_abs': 'C',
```

Building the final parser

We have now three kinds of parsers:

`*` variableParser : `handle variables names``*` paramParser : `handle a set of variable definitions``*` tableColParser : `handle tables defined column by column`

There are two more in `ConfigNumParser` :

`*` tableRowParser : `handle tables defined row by row``*` matrixParser : `handle matrix containg only python values or NaN`

We won't detail them here, because they use exactly the same techniques we've already seen.

We will rather see how to combine them into a complex parser as it is done in the `parseConfigFile` function:

```
# Section header
sectionName = Suppress('[') + keyName + Suppress(']')

# Group section name and content
section = Group (sectionName +
                  ( paramParser()
                  | tableColParser()
                  | tableRowParser()
                  | matrixParser()
            )     )

# Build the final parser and suppress empty sections
parser = Dict( OneOrMore( section | Suppress(sectionName) ))

# Defines comments
parser.ignore('#' + restOfLine)
```

That's all.

The parser can now be use through its method `parseString` or `parseFile` . See [attachment:ConfigNumParser_v0.1.1.py ConfigNumParser] for more information.

I hope this will give you a good starting point to read complex formatted text.

Attachments

- `ConfigNumParser_v0.1.1.py`
- `ConfigNumParser_v0.1.py`
- `data.txt`
- `data3.txt`

Scripting Mayavi 2: basic modules

Introduction

These modules are called “basic” because they are general and independant to all kind of data.

Before using a module, remind that you have to import it.

In general, if you want to change the color of an object, you have to type:

```
module.actor.property.color = fg_color
```

where fg_color is a “tuple”, say (0, 0, 0) for black.

Note: You can set color for each module/filter (when available, of course), as above. But you can also set background/foreground colors for all your !MayaVi2 sessions. See [:Cookbook/MayaVi/Tips: Cookbook/MayaVi/Tips]. So, your background and foreground colors may (in fact, must) be different from those in the pictures presented here.

Outline module

Nothing special for this very simple module.

Begin to import the Outline module:

```
from enthought.mayavi.modules.outline import Outline
```

then

```
fg_color = (0, 0, 0)    # black foreground color
o = Outline()
script.add_module(o)
o.actor.property.color = fg_color
```

[](files/attachments/MayaVi_ScriptingMayavi2_BasicModules/basic_outline.png

Axes module

For axis, you can set several parameters:

```
* color for axes;``* color for labels;``* labels name;``* and label
```

Here’s what you can type:

```
from enthought.mayavi.modules.axes import Axes
```

then

```
a = Axes()
script.add_module(a)
a.axes.property.color = fg_color                     # color for ax
a.axes.axis_title_text_property.color = fg_color     # color for ax
a.axes.x_label = "Lx"                                # label for Ox
a.axes.y_label = "Ly"                                # label for Oy
a.axes.z_label = "Lz"                                # label for Oz
a.axes.label_format = ""                             # no dimension
a.axes.axis_label_text_property.color = fg_color     # in case we w
```

Label format is the format for the dimensions along Ox, Oy and Oz axis. By default, it is set to %6.3g.

[](files/attachments/MayaVi_ScriptingMayavi2_BasicModules/basic_axes.png

OrientationAxes module

!OrientationAxes module will display a little trihedron in the corner of the render window, showing the orientation of the three axes, Ox, Oy, Oz.

Note: VTK > 5.0 must be installed in order to use this module.

Nothing special here, as the Outline module, you can set the labels color:

```
from enthought.mayavi.modules.orientation_axes import OrientationAx
```

and

```
oa = OrientationAxes()
script.add_module(oa)
oa.text_property.color = fg_color
```

[]
(files/attachments/MayaVi_ScriptingMayavi2_BasicModules/basic_orientationaxes
.png

Text module

You can use this module to display a title for your rendering window.

You need a text (a string), and you have to set up its position in the window (coordinates go from 0 to 1 in x and y) with the x_position and y_position parameters.

Then you can set up the height and the width of your text:

```
from enthought.mayavi.modules.text import Text
```

and

```
# I like my title centered and at top of the window
t = Text()
t.text = "My Title Centered at the Top"
script.add_module(t)
t.actor.scaled_text = False
t.actor.text_property.font_size = 34
t.actor.text_property.color = fg_color
# have to find out text width to center it. Tricky, but works fine
t.width = 1.0*t.actor.mapper.get_width(t.scene.renderer)/t.scene.re
height = 1.0*t.actor.mapper.get_height(t.scene.renderer)/t.scene.re
t.x_position = 0.5-t.width/2
t.y_position = 1-height
```

[](files/attachments/MayaVi_ScriptingMayavi2_BasicModules/basic_title.png

Now, we will present how to set up the color bar, called “scalar” or “vector” color bar. It depends of the data you have in your file.

Setting up color bar

Strictly speaking, the color bar is not a module, i.e. you don’t need to add it with an add_module() command: you have to associate the “module_manager” object to a module, previously loaded, say the Text module for example.

Then, you can configure the color bar as follows (keywords are self-explanatory):

```
mmsclut = t.module_manager.scalar_lut_manager
mmsclut.show_scalar_bar = True
mmsclutsc = mmsclut.scalar_bar
mmsclutsc.orientation = "vertical"    # or "horizontal"
mmsclutsc.width = 0.1
mmsclutsc.height = 0.8
mmsclutsc.position = (0.01, 0.15)     # color bar located to the le
mmsclutsc.label_text_property.color = fg_color
mmsclutsc.title_text_property.color = fg_color
mmsclut.number_of_labels = 10
mmsclut.number_of_colors = 64
mmsclut.data_name = "My Label"
```

[](files/attachments/MayaVi_ScriptingMayavi2_BasicModules/basic_colorbar.png

Note: To configure a color bar for vectors instead of scalars, replace “scalar_lut_manager” by “vector_lut_manager” above.

At last, to close the “basic” modules section, let’s see how we can setting up the scene.

Setting up the scene

By “setting up the scene”, you have to read “how the scene will be seen”: for example, setting the color background and the point of view of the scene.

As usual, setting these parameters using python & TVTK is very easy.

If you want to change background color, you may need to also change foreground color for all modules. We recall them here.

```
#! /usr/bin/env python

from enthought.mayavi.modules.outline import Outline
from enthought.mayavi.modules.axes import Axes
from enthought.mayavi.modules.orientation_axes import OrientationAx
from enthought.mayavi.modules.text import Text

# we want a dark foreground color on a bright background
fg_color = (0.06666, 0.06666, 0.1804)   # dark blue
bg_color = (1, 1, 0.94118)              # ivory

# setting foreground color for Outline module
o = Outline()
script.add_module(o)
o.actor.property.color = fg_color

# setting foreground color for Axes module
```

```
a = Axes()
script.add_module(a)
a.axes.property.color = fg_color                     # color for ax
a.axes.axis_title_text_property.color = fg_color     # color for ax
a.axes.x_label = "Lx"                                # label for Ox
a.axes.y_label = "Ly"                                # label for Oy
a.axes.z_label = "Lz"                                # label for Oz
a.axes.label_format = ""                             # no dimensior

# setting foreground color for OrientationAxes module
oa = OrientationAxes()
script.add_module(oa)
oa.text_property.color = fg_color

# setting foreground color for Text module
t = Text()
t.text = "My Title Centered at the Top"
script.add_module(t)
t.actor.scaled_text = False
t.actor.text_property.font_size = 34
t.actor.text_property.color = fg_color
t.width = 1.0*t.actor.mapper.get_width(t.scene.renderer)/t.scene.re
height = 1.0*t.actor.mapper.get_height(t.scene.renderer)/t.scene.re
t.x_position = 0.5-t.width/2
t.y_position = 1-height

# setting foreground color for labels and title color bar.
mmsclut = t.module_manager.scalar_lut_manager
mmsclut.show_scalar_bar = True
mmsclutsc = mmsclut.scalar_bar
mmsclutsc.orientation = "vertical"
mmsclutsc.width = 0.1
mmsclutsc.height = 0.8
mmsclutsc.position = (0.01, 0.15)
mmsclutsc.label_text_property.color = fg_color
mmsclutsc.title_text_property.color = fg_color
mmsclut.number_of_labels = 10
mmsclut.number_of_colors = 64
mmsclut.data_name = "My Label"

# setting background color for the scene.
t.scene.background = bg_color
```

Some points of view are also predefined in !MayaVi2.

If you want:

```
* Ox axis normal to the scene: use x_plus_view() (towards) or x_minu
```

```
* Oy axis normal to the scene: use y_plus_view() (towards) or y_minu
```

```
* Oz axis normal to the scene: use z_plus_view() (towards) or z_minu
```

```
* an isometric view (coordinates normal are (1, 1, 1)), use isometri
```

You can also:

```
* set the elevation and azimuth angles to your needs (in degrees);
```

```
* set a zooming factor of your scene.
```

with:

```
t.scene.x_plus_view()
t.scene.camera.azimuth(62)
t.scene.camera.elevation(19.5)
t.scene.camera.zoom(1.5)
```

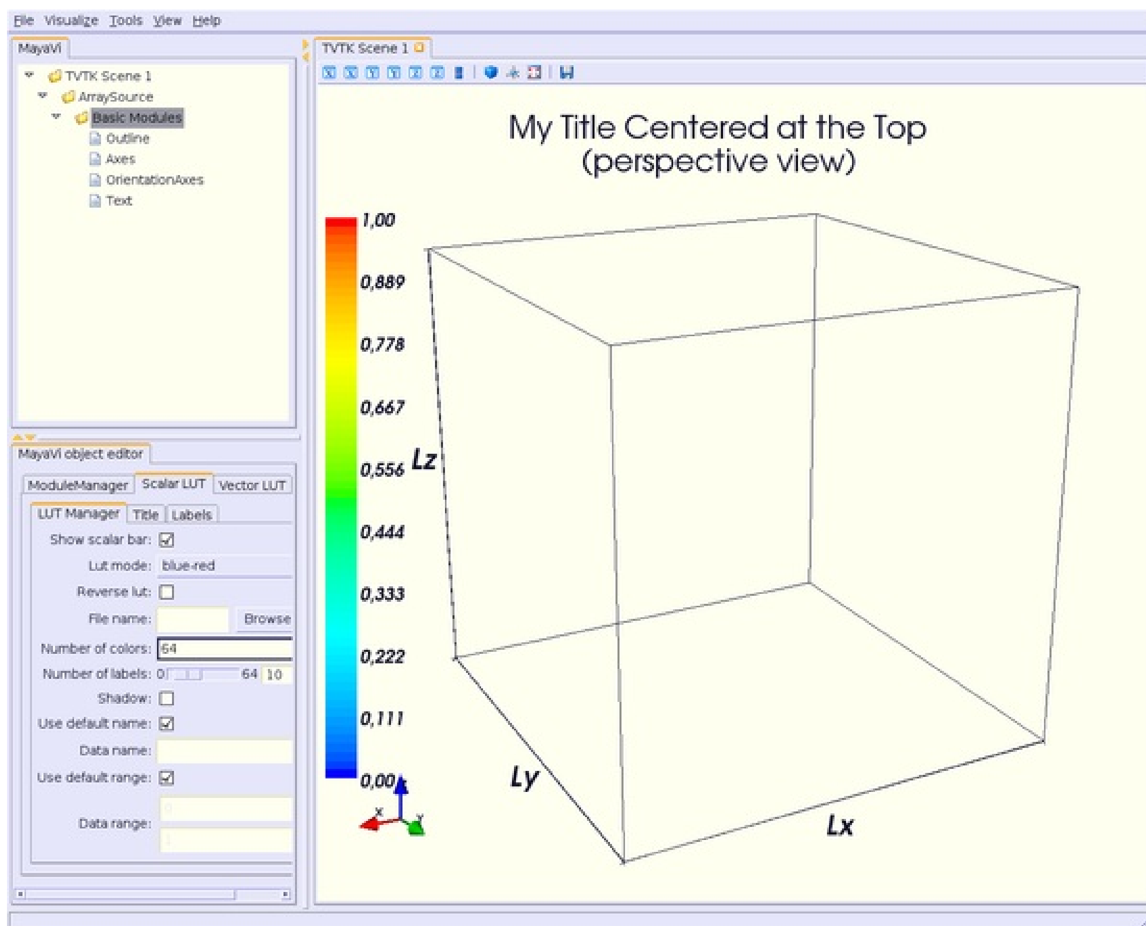
At last, you can choose if you want a perspective view or a parallel projection for your scene:

```
t.scene.camera.parallel_projection = True
```

[]
(files/attachments/MayaVi_ScriptingMayavi2_BasicModules/basic_scene_parall.png

for a parallel projection, or:

```
t.scene.camera.parallel_projection = False
```



for a perspective view.

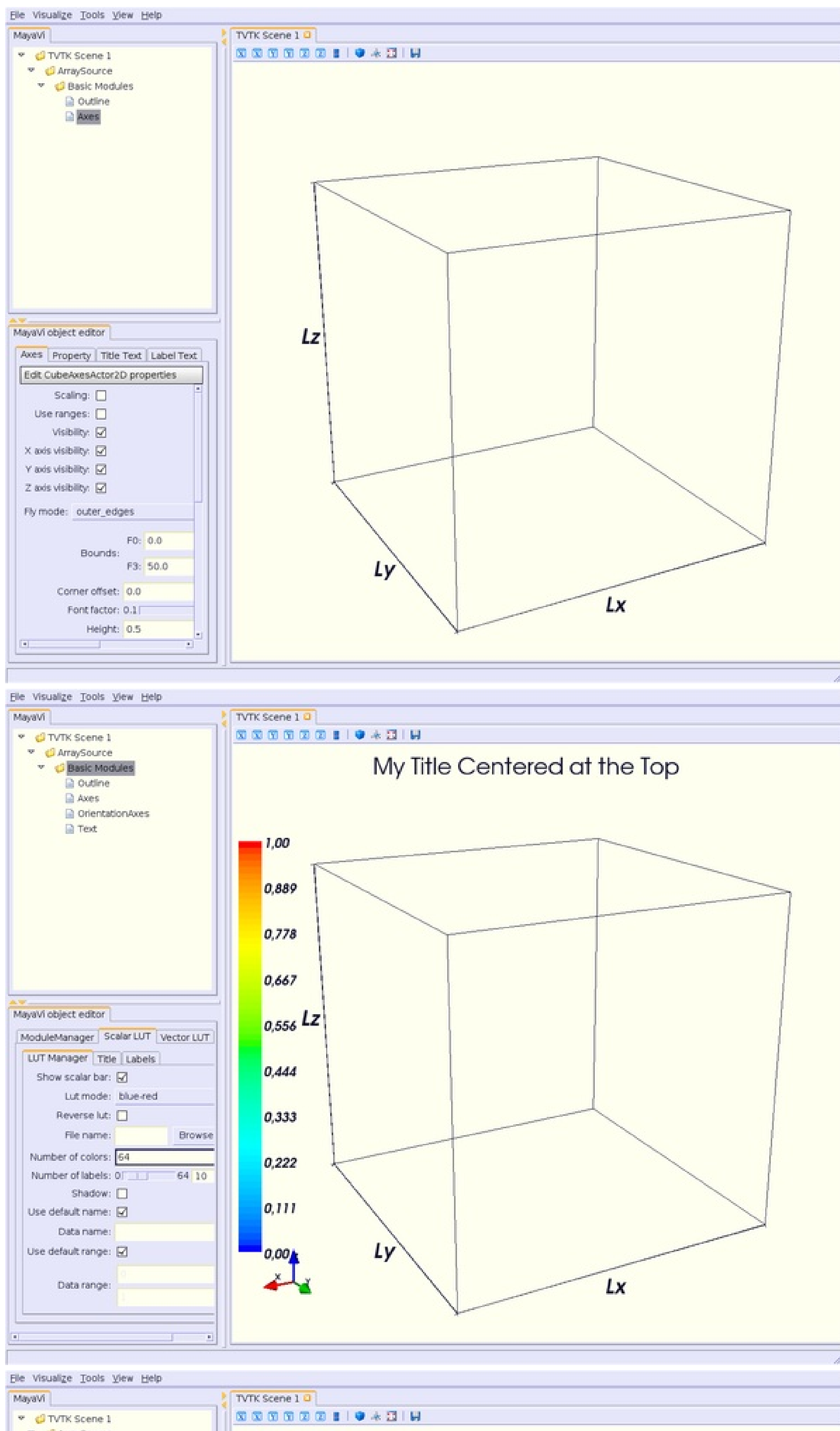
Here, “t” stands for the Text module previously loaded.

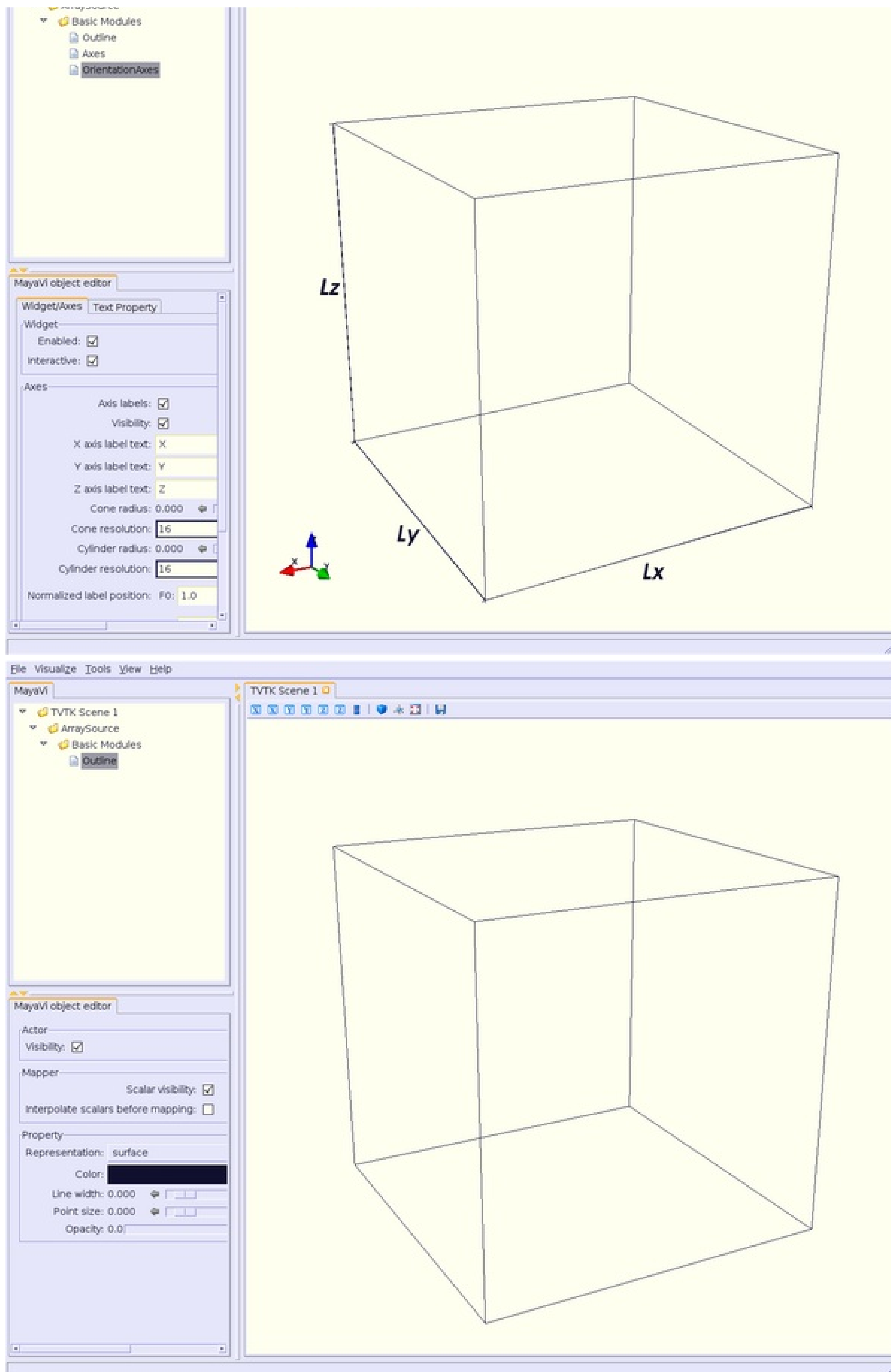
Note: There are a lot of others parameters you can set up for your scene. See [:Cookbook/MayaVi/Tips: Cookbook/MayaVi/Tips] to read how to get more information about setting parameters modules.

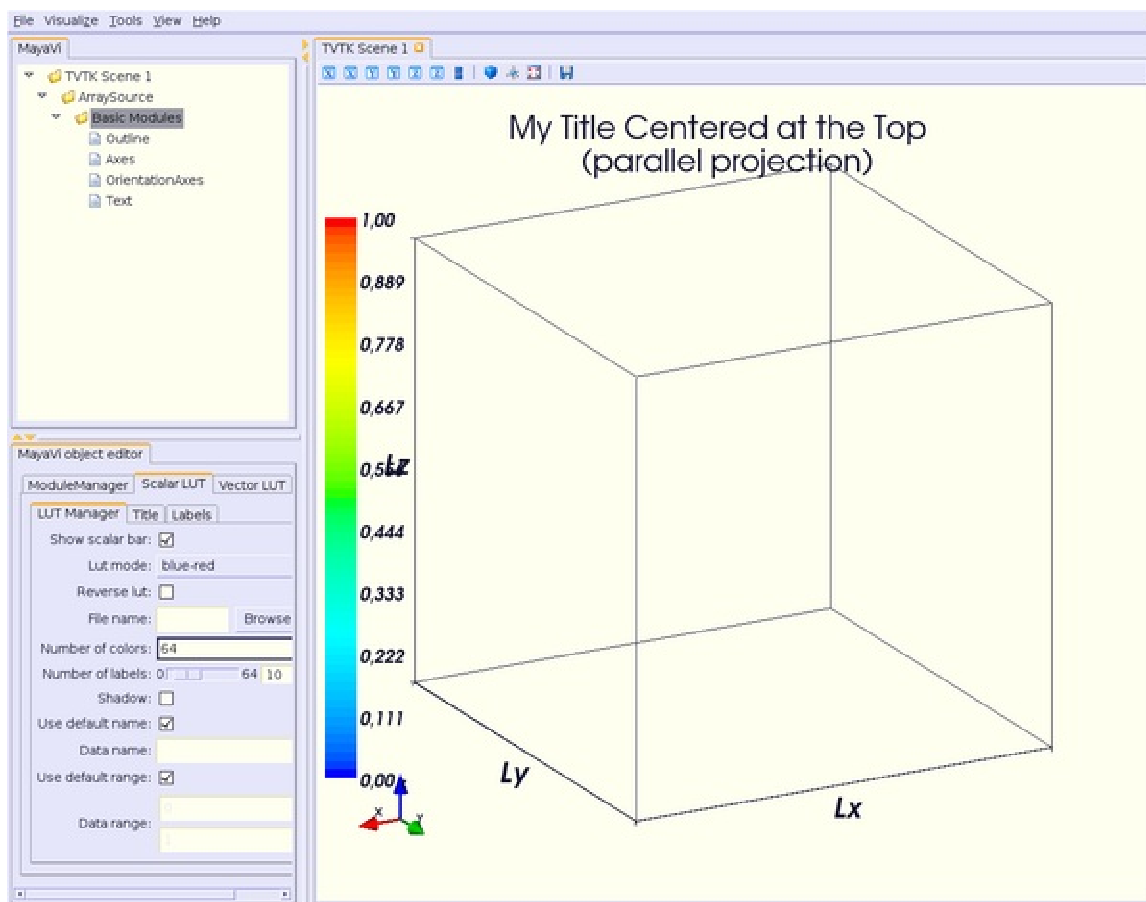
Now, it’s time to read the most interesting part: configuring and using modules and filters which interact with your data.

Attachments

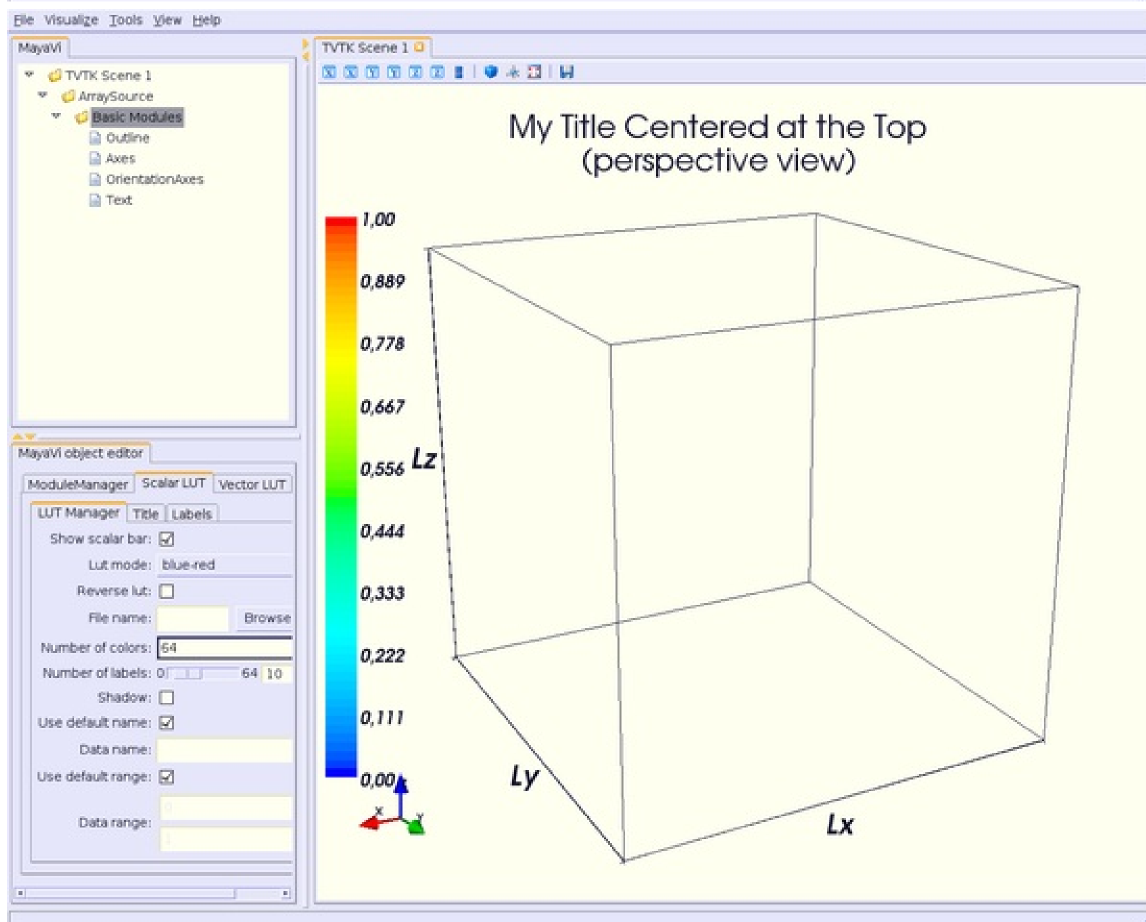
- `basic_axes.png`
- `basic_colorbar.png`
- `basic_orientationaxes.png`
- `basic_outline.png`
- `basic_scene_parall.png`
- `basic_scene_persp.png`
- `basic_title.png`


File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
MayaVi object editor
Axes
Property
Title Text
Label Text
Edit CubeAxesActor2D properties
Scaling:
Use ranges:
Visibility:
X axis visibility:
Y axis visibility:
Z axis visibility:
Fly mode: outer_edges
Bounds:
F0: 0.0
F3: 50.0
Corner offset: 0.0
Font factor: 0.1
Height: 0.5
Lz
Ly
Lx
OrientationAxes
Text
ModuleManager
Scalar LUT
Vector LUT
LUT Manager
Title
Labels
Show scalar bar:
Lut mode: blue-red
Reverse lut:
File name:
Browse
Number of colors: 64
Number of labels: 0 64 10
Shadow:
Use default name:
Data name:
Use default range:
Data range:
My Title Centered at the Top
1,00
0,889
0,778
0,667
0,556
0,444
0,333
0,222
0,111
0,00

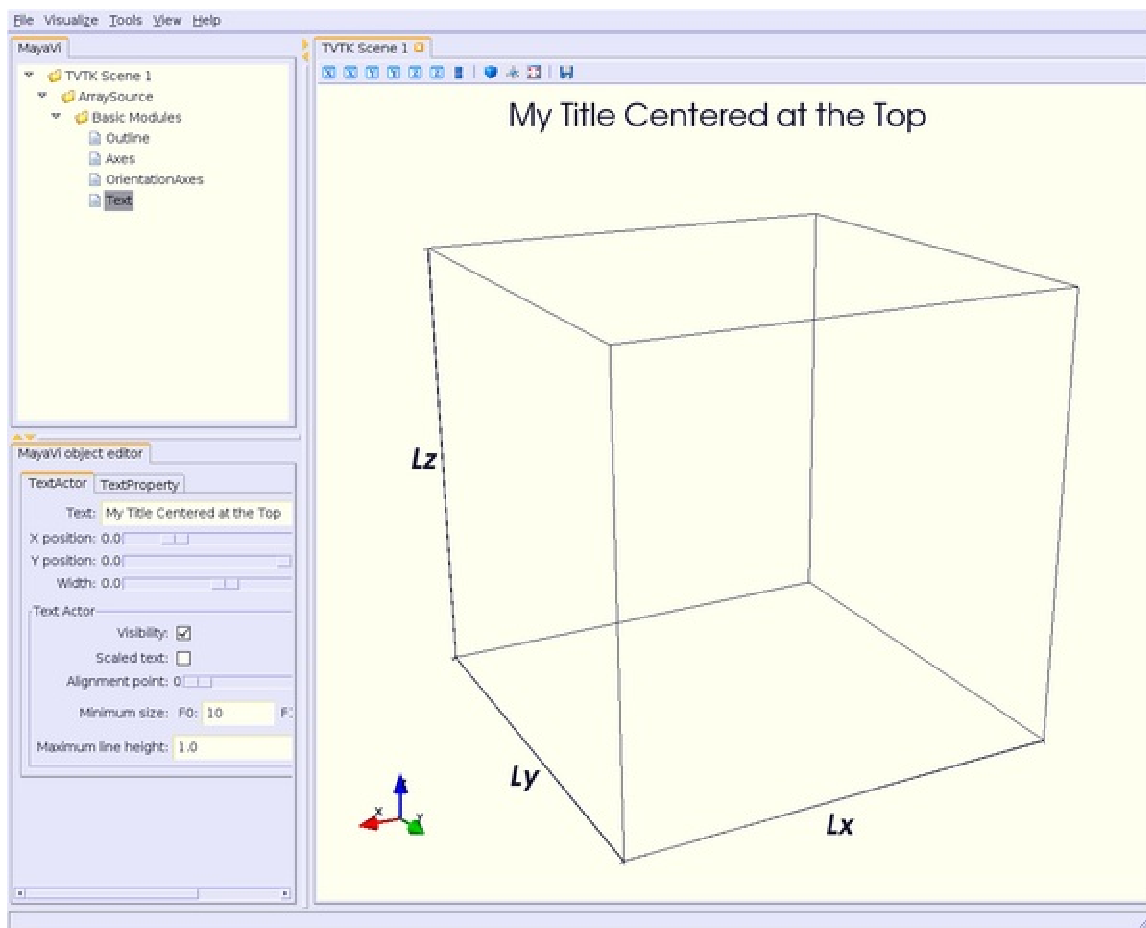

Basic Modules
Outline
Axes
OrientationAxes
Lz
Ly
Lx
MayaVi object editor
Widget/Axes
Text Property
Widget
Enabled:
Interactive:
Axes
Axis labels:
Visibility:
X axis label text:
Y axis label text:
Z axis label text:
Cone radius: 0.000
Cone resolution: 16
Cylinder radius: 0.000
Cylinder resolution: 16
Normalized label position: F0: 1.0
File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
MayaVi object editor
Actor
Visibility:
Mapper
Scalar visibility:
Interpolate scalars before mapping:
Property
Representation: surface
Color:
Line width: 0.000
Point size: 0.000
Opacity: 0.0



File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
MayaVi object editor
ModuleManager
Scalar LUT
Vector LUT
LUT Manager
Title
Labels
Show scalar bar:
Lut mode:
blue-red
Reverse lut:
File name:
Browse
Number of colors:
64
Number of labels:
Shadow:
Use default name:
Data name:
Use default range:
Data range:
TVTK Scene 1
My Title Centered at the Top
(parallel projection)
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111
0,00
Lz
Ly
Lx



File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
MayaVi object editor
ModuleManager
Scalar LUT
Vector LUT
LUT Manager
Title
Labels
Show scalar bar:
Lut mode:
blue-red
Reverse lut:
File name:
Browse
Number of colors:
64
Number of labels:
Shadow:
Use default name:
Data name:
Use default range:
Data range:
TVTK Scene 1
My Title Centered at the Top
(perspective view)
1,00
0,889
0,778
0,667
0,556
0,444
0,333
0,222
0,111
0,00
Lz
Ly
Lx


File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
MayaVi object editor
TextActor
TextProperty
Text: My Title Centered at the Top
X position: 0.0
Y position: 0.0
Width: 0.0
Text Actor
Visibility:
Scaled text:
Alignment point: 0
Minimum size: F0: 10
Maximum line height: 1.0
TVTK Scene 1
My Title Centered at the Top
Lz
Ly
Lx

Scripting Mayavi 2: filters

Introduction

Well, until now, examples above are quite simple: scalars or vectors data are presented in the “vaccum”, i.e. there is not object or material or whatsoever.

In others words, how can one display some object, say a metallic parallellepiped, for example, in a scalars or vectors field ?

The first filter you will be presented here deals with this problem.

ExtractUnstructuredGrid filter

For this example, we suppose several hypotheses:

```
* the mesh of the volume data is made of 76x60x72 cubic cells (8 ve
```

```
* a metallic parallellepiped is immersed in a EM field spreading ove
```

```
* as there are different kinds of cells, the !UnstructuredGrid data
```

<http://www.vtk.org/pdf/file-formats.pdf> `__` to know how !UnstructuredGrid files must be written, VTK cells syntax, etc).`

To display the metallic parallellepiped as a separate object from the vaccum, you have to extract the cells corresponding to this object. Thus, you will be able to display this object using the Surface module for example.

First, import !ExtractUnstructuredGrid filter and Surface module, as usual:

```
from enthought.mayavi.modules.surface import Surface

from enthought.mayavi.filters.extract_unstructured_grid import Extı
```

then

```
### for the metallic parallellepiped
script.engine.current_object = src  # current object must be set to
eug1 = ExtractUnstructuredGrid()
script.add_filter(eug1)
eug1.filter.cell_clipping = True
eug1.filter.cell_minimum = 342881
eug1.filter.cell_maximum = 345966
s = Surface()                       # the metallic is displayed usi
eug1.add_module(s)                  # the module must be added to t
s.actor.mapper.scalar_visibility = False
s.actor.property.color = (0.509804, 0.5098040, 0.5490196) # grey co

### we need also extract the required cells for and only for the va
script.engine.current_object = src  # current object must be set to
eug2 = ExtractUnstructuredGrid()
script.add_filter(eug2)
eug2.filter.cell_clipping = True
eug2.filter.cell_minimum = 0
eug2.filter.cell_maximum = 342880

### here, we can display the EM field using ScalarCutPlane/VectorCu
### Surface, Vectors modules as usual
.../...
```

This should look like this:

[](files/attachments/MayaVi_ScriptingMayavi2_Filters/filter_eug1.png)

For this first example, there was only one object, and it was faceted.

Now, say we have a second object, not metallic but dielectric (so the EM field within it should not be null). Thus we have to use some 3D cells, as VTK*HEXAHEDRON cells (cell ids go from #345967 to #349094). We also want to display the field on the surface on the metallic object _and* in the dielectric object.

```
### for the metallic parallellepiped
script.engine.current_object = src
eug1 = ExtractUnstructuredGrid()
script.add_filter(eug1)
eug1.filter.cell_clipping = True
eug1.filter.cell_minimum = 342881
eug1.filter.cell_maximum = 345966
s = Surface()
eug1.add_module(s)
s.actor.mapper.scalar_visibility = True # scalar field on the surfa

### for the dielectric parallellepiped
script.engine.current_object = src
eug2 = ExtractUnstructuredGrid()
script.add_filter(eug2)
eug2.filter.cell_clipping = True
eug2.filter.cell_minimum = 345967
eug2.filter.cell_maximum = 349094
s = Surface()
eug2.add_module(s)
s.actor.mapper.scalar_visibility = True # scalar field set to on
s.enable_contours = True                # in the volume

### we need also extract the required cells for and only for the va
script.engine.current_object = src  # current object must be set to
eug3 = ExtractUnstructuredGrid()
script.add_filter(eug3)
eug3.filter.cell_clipping = True
eug3.filter.cell_minimum = 0
eug3.filter.cell_maximum = 342880

.../...
```

This should render this:

[](files/attachments/MayaVi_ScriptingMayavi2_Filters/filter_eug2.png

ExtractGrid filter

Using !ExtractGrid filter is easier, because it works (only) on structured grids: you only have to set min/max values for x, y, z coordinates. Thus, you can cut a subvolume of your data. You can also apply a ratio on each coordinates, to decrease the cells number.

Import, as usual, the required modules and/or filter:

```
from enthought.mayavi.modules.surface import Surface

from enthought.mayavi.filters.extract_grid import ExtractGrid
```

then you can set filter’s limits as:

```
eg = ExtractGrid()
script.add_filter(eg)
eg.x_min, eg.x_max = 10, 40
eg.y_min, eg.y_max = 10, 40
eg.z_min, eg.z_max = 10, 40

# eg.x_ratio = 2
# eg.y_ratio = 2
# eg.z_ratio = 2

# same example using Surface module
s = Surface()
s.enable_contours = True
s.contour.auto_contours = True
s.contour.number_of_contours = 10
s.actor.property.opacity = 0.2
script.add_module(s)
s.contour.minimum_contour = 0
s.contour.maximum_contour = 1
s.module_manager.scalar_lut_manager.data_range = [0, 1]
```

[](files/attachments/MayaVi_ScriptingMayavi2_Filters/filter_eg.png

Threshold filter

Using this filter, you can consider scalars values contained in a specific range.

Suppose that your scalars data spread from 0 to 1, but you are only interested by the values in the range [0.4, 0.6] and you want to play with the slidebar of the !IsoSurface module within this range, around 0.5. By default, min & max values of the slidebar will be set to 0 & 1, because of your data range:

[](files/attachments/MayaVi_ScriptingMayavi2_Filters/filter_thrld1.png

To play more accurately with the slidebar of the !IsoSurface module, you have to set min & max values to the required values, i.e. 0.4 & 0.6. Thus, if you want to see your scalars data around 0.5, you can set the slidebar from 0.4 to 0.6 more easily than in the case where slidebar goes form 0 to 1.

The Threshold filter can help you to do this.

Begin to import the module and the filter first:

```
from enthought.mayavi.modules.iso_surface import IsoSurface

from enthought.mayavi.filters.threshold import Threshold
```

then, set the threshold values:

```
thh = Threshold()
script.add_filter(thh)
thh.lower_threshold = 0.4
thh.upper_threshold = 0.6
isosurf = IsoSurface()
thh.add_module(isosurf)
isosurf.contour.contours = [0.5]
isosurf.compute_normals = True
isosurf.actor.property.opacity = 0.2
isosurf.module_manager.scalar_lut_manager.data_range = [0, 1]
```

and you’re done !

This should look like this:

[](files/attachments/MayaVi_ScriptingMayavi2_Filters/filter_thrld2.png

You can notice on the two previous figures that the Threshold module approximates bounds to the nearest values (there are not strictly equal to 0.4 & 0.6).

PointToCellData filter

Generally, data are interpolated between each point. Thus, they look like nicer.

But maybe in some case, you don’t want them to be interpolated, and see the data “as they are”: they are not displayed as points, but as cells. In this case, you can use the !PointToCellData filter.

Let’s see again the example using the !ScalarCutPlane module, and import the !PointToCellData filter:

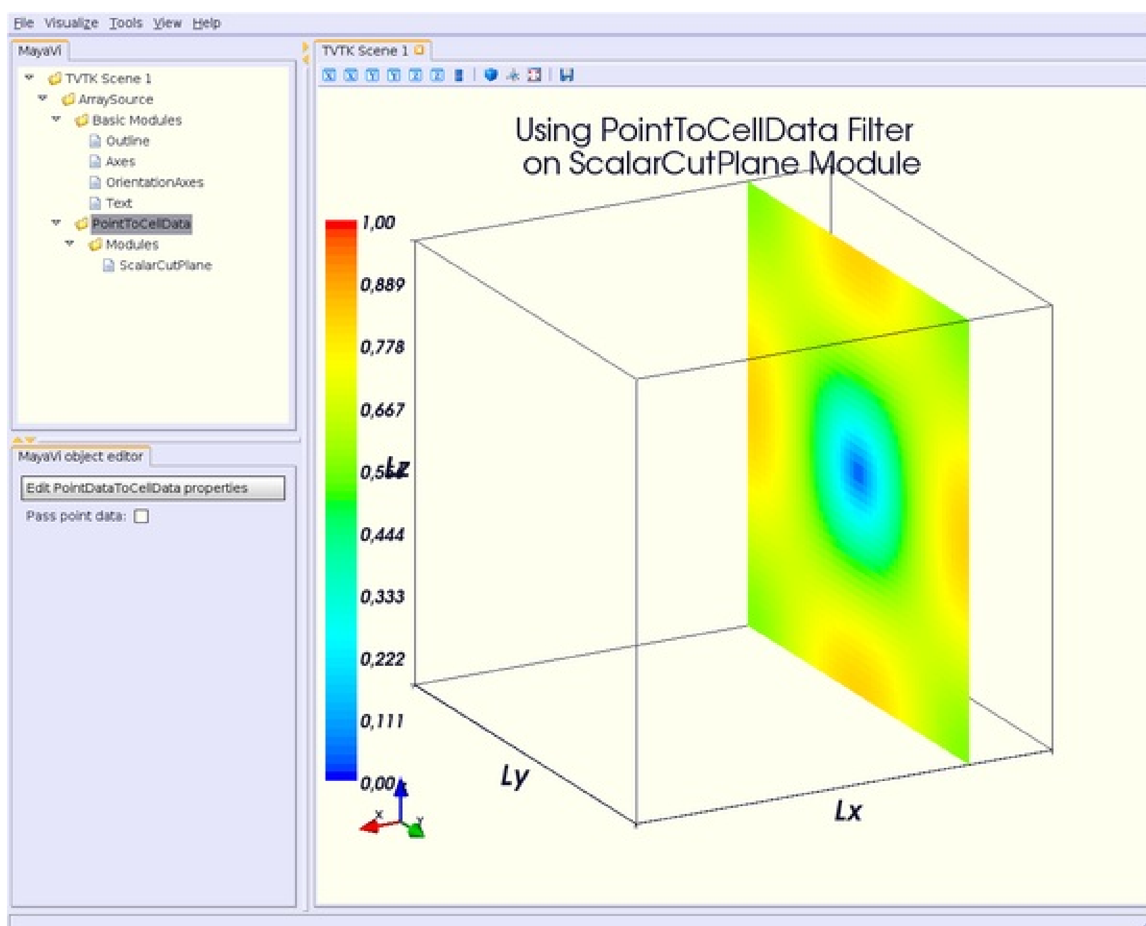
```
from enthought.mayavi.modules.scalar_cut_plane import ScalarCutPlan

from enthought.mayavi.filters.point_to_cell_data import PointToCell
```

then add the !ScalarCutPlane module “above” the !PointToCellData filter, as usual:

```
ptocd = PointToCellData()
script.add_filter(ptocd)
scp = ScalarCutPlane()
ptocd.add_module(scp)
scp.implicit_plane.normal = (1, 0, 0)
scp.implicit_plane.origin = (10, 25, 25)
scp.implicit_plane.widget.enabled = False
scp.actor.property.diffuse = 0.0
scp.actor.property.ambient = 1.0
scp.actor.property.opacity = 1.0
scp.module_manager.scalar_lut_manager.data_range = [0, 1]
```

Thus, you can see your data on each cell and not as points (compare to the first figure showing the use of the !ScalarCutPlane module):



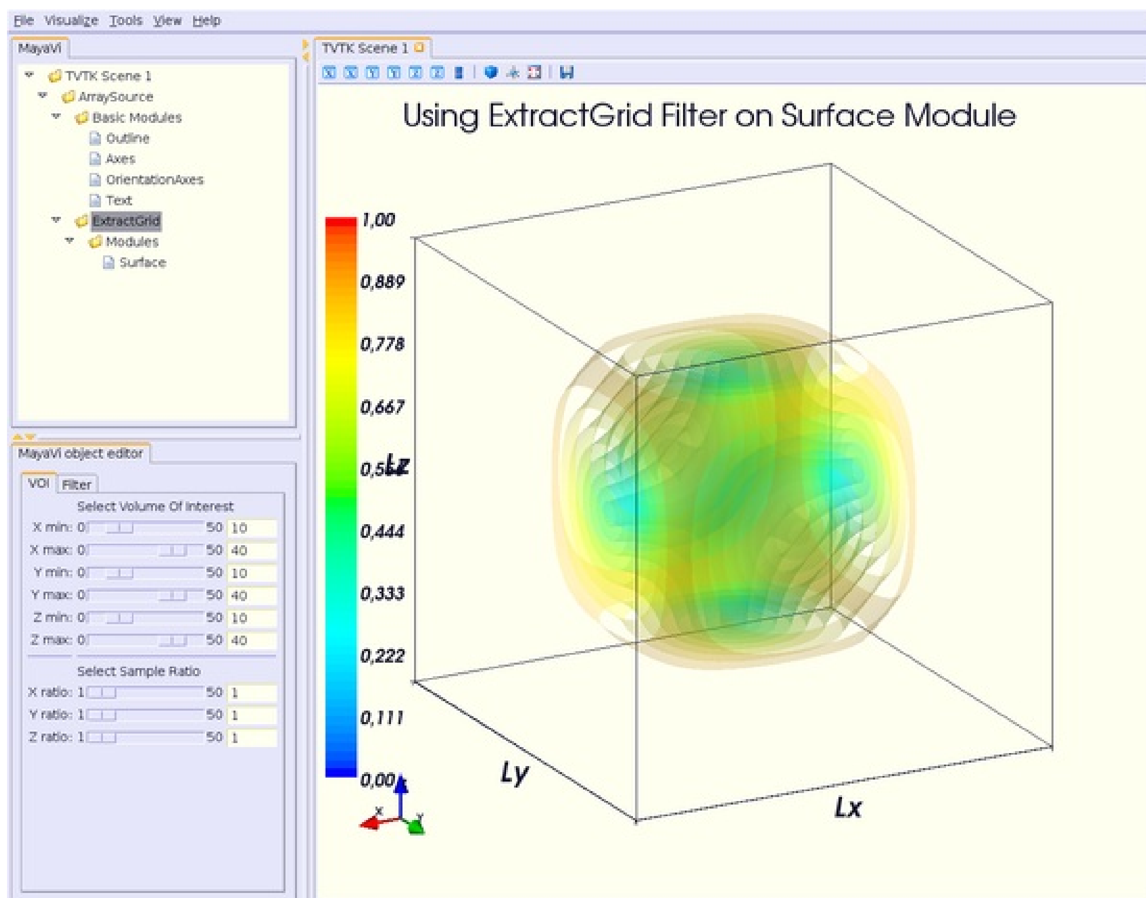
WarpScalar filter

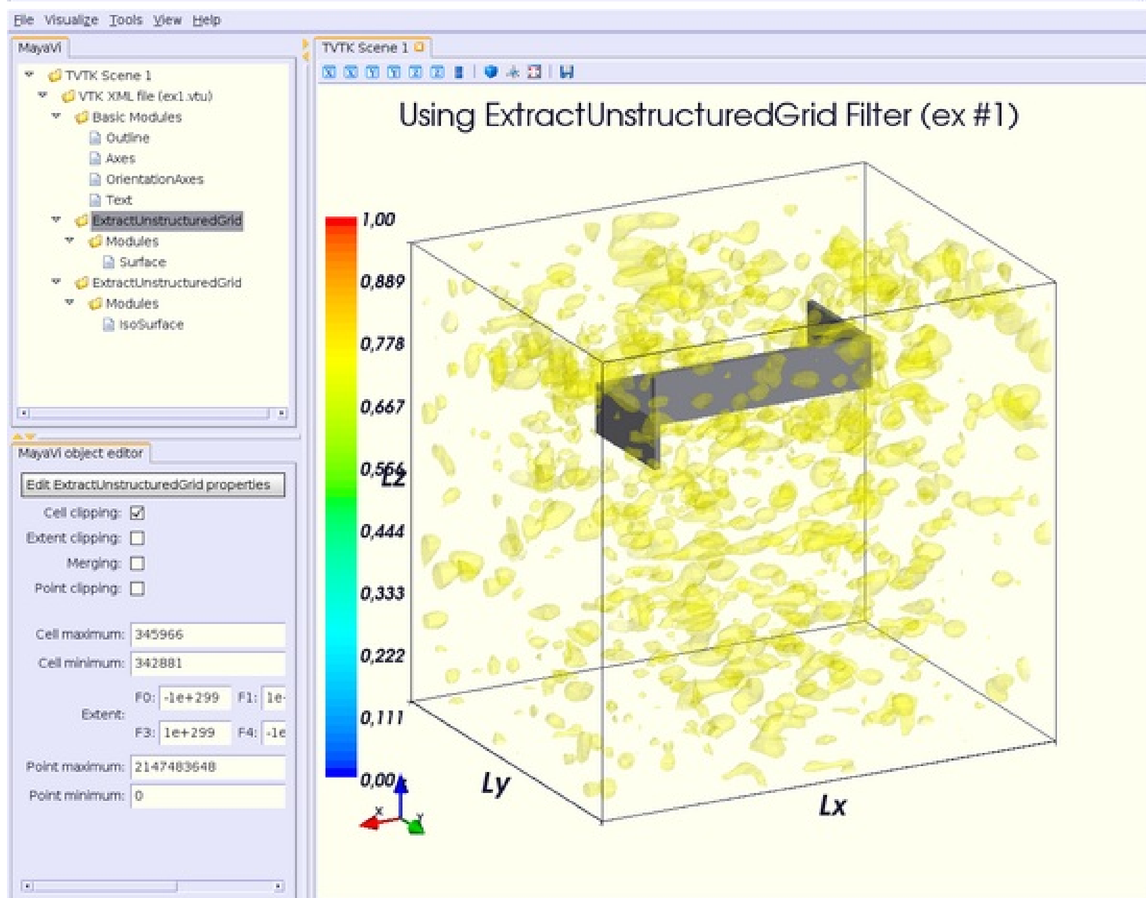
You can use the !WarpScalar filter to warp a 2D surface for example. See [:Cookbook/MayaVi/Examples: Example using mlab (surf_regular_mlab.py)].

TransformData filter

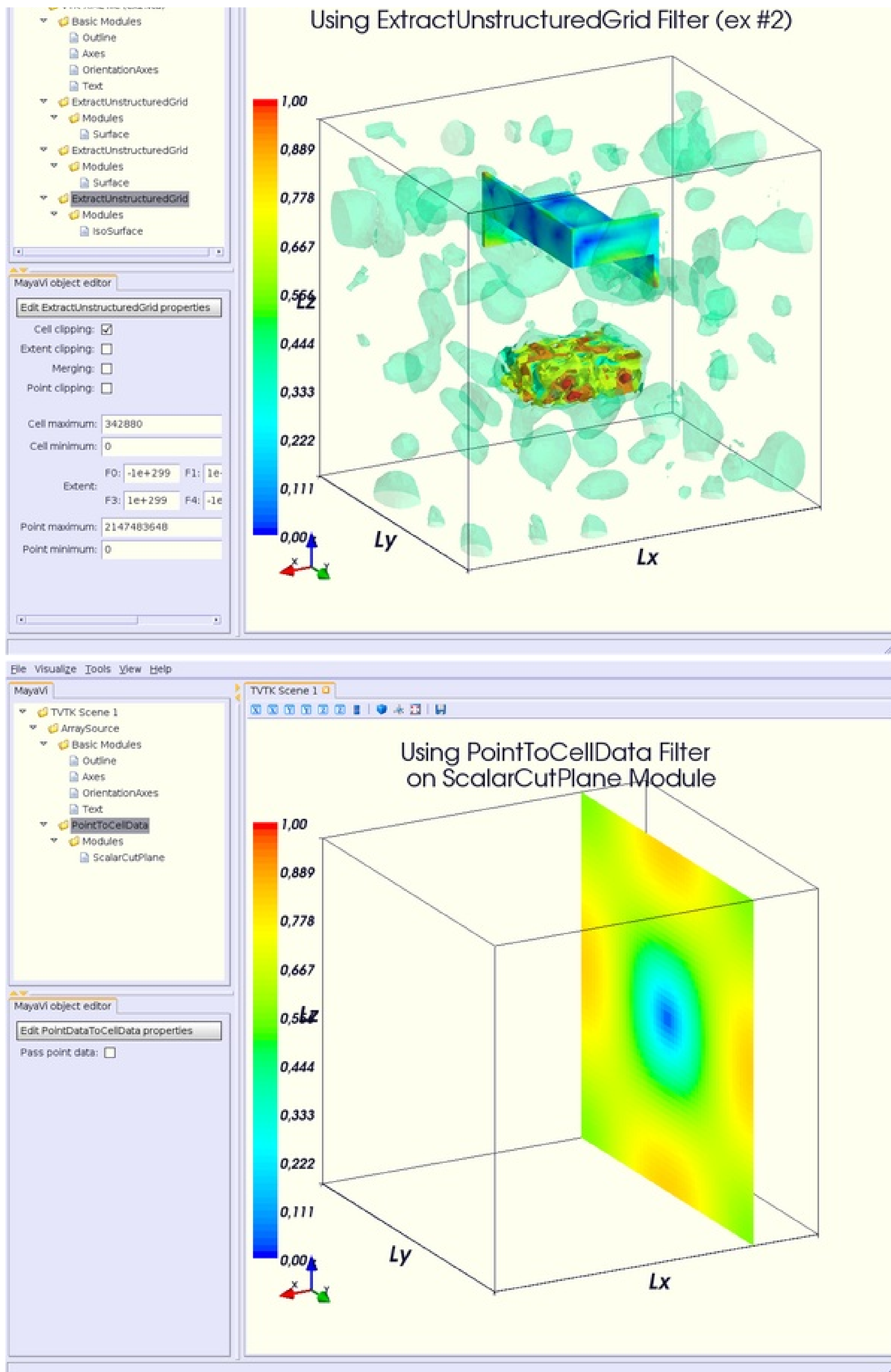
Attachments

- `filter_eg.png`
- `filter_eug1.png`
- `filter_eug2.png`
- `filter_p2c.png`
- `filter_thrld1.png`
- `filter_thrld2.png`

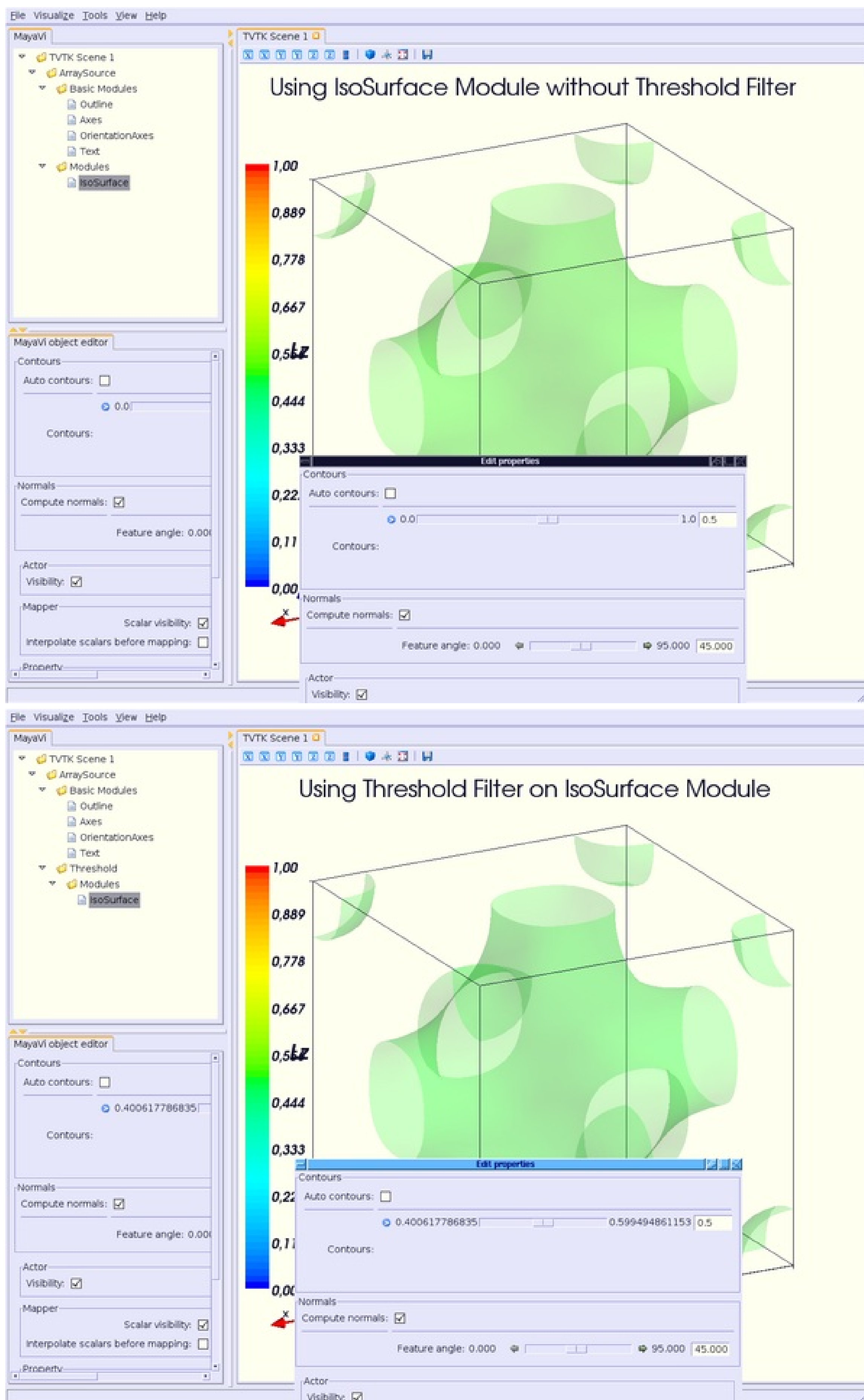

File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
ExtractGrid
Modules
Surface
MayaVi object editor
VOI
Filter
Select Volume Of Interest
X min:
X max:
Y min:
Y max:
Z min:
Z max:
Select Sample Ratio
X ratio:
Y ratio:
Z ratio:
Using ExtractGrid Filter on Surface Module
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111
Ly
Lx


File Visualize Tools View Help
MayaVi
TVTK Scene 1
VTK XML file (ex1.vtu)
Basic Modules
Outline
Axes
OrientationAxes
Text
ExtractUnstructuredGrid
Modules
Surface
ExtractUnstructuredGrid
Modules
IsoSurface
MayaVi object editor
Edit ExtractUnstructuredGrid properties
Cell clipping:
Extent clipping:
Merging:
Point clipping:
Cell maximum: 345966
Cell minimum: 342881
Extent:
F0: -1e+299
F3: 1e+299
Point maximum: 2147483648
Point minimum: 0
Using ExtractUnstructuredGrid Filter (ex #1)
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111
Ly
Lx


File Visualize Tools View Help
MayaVi
TVTK Scene 1
VTK XML file (ex2.vtu)



Using ExtractUnstructuredGrid Filter (ex #2)
Basic Modules
Outline
Axes
OrientationAxes
Text
ExtractUnstructuredGrid
Modules
Surface
ExtractUnstructuredGrid
Modules
Surface
ExtractUnstructuredGrid
Modules
IsoSurface
MayaVi object editor
Edit ExtractUnstructuredGrid properties
Cell clipping:
Extent clipping:
Merging:
Point clipping:
Cell maximum: 342880
Cell minimum: 0
Extent:
Point maximum: 2147483648
Point minimum: 0
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111
Lz
Ly
Lx
File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
PointToCellData
Modules
ScalarCutPlane
MayaVi object editor
Edit PointDataToCellData properties
Pass point data:
Using PointToCellData Filter
on ScalarCutPlane Module
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111
Lz
Ly
Lx



File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
Modules
IsoSurface
Using IsoSurface Module without Threshold Filter
1,00
0,889
0,778
0,667
0,444
0,333
MayaVi object editor
Contours
Auto contours:
Contours:
Normals
Compute normals:
Actor
Visibility:
Mapper
Scalar visibility:
Interpolate scalars before mapping:
Edit properties
0.0
1.0
0.5
Feature angle: 0.000
95.000
45.000
File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
Threshold
Modules
IsoSurface
Using Threshold Filter on IsoSurface Module
1,00
0,889
0,778
0,667
0,444
0,333
MayaVi object editor
Contours
Auto contours:
0.400617786835
Contours:
Normals
Compute normals:
Actor
Visibility:
Mapper
Scalar visibility:
Interpolate scalars before mapping:
Edit properties
0.400617786835
0.599494861153
0.5
Feature angle: 0.000
95.000
45.000

Scripting Mayavi 2: main modules

Introduction

Here's the real stuff ;-)

You'll be learned here how you can use the various modules in !MayaVi2.

Note: Some modules can't be added for any type of data set. Some work only for !StructuredGrid or !StructuredPoints for example (see http://www.vtk.org/pdf/file-formats.pdf for more information about VTK data type). It will be specified each time is needed.

Note2: In the !MayaVi2 tree view, the "Main Modules" (called "Modules") have been separated from the "Basic Modules" loading the !ModuleManager. Of coutse, you can load all your modules and filters without using the !ModuleManager.

ImagePlaneWidget/ScalarCutPlane/SliceUnstructuredGrid module

The simpliest (and easiest, but not the most impressive ;-)) way to display 3D data is doubtless to slice it on some planes, normal to Ox, Oy or Oz axis, or oblique.

One of the modules you can use to do this is called !ScalarCutPlane. It works for any data.

Note: As the !ImagePlaneWidget module also display scalars data on a plane (but it does not "cut" data), please see [:Cookbook/MayaVi/Examples: Example with a 3D array as numerical source (numeric_source.py)] or [:Cookbook/MayaVi/Examples: Example using ImagePlaneWidget Module (test.py)] to get more information on how you can use this module.

You have to set several parameters:

```
* plane normal;``* its origin;``* widget enabled or not: if enabled,
```

Thus, you have to type:

```
from enthought.mayavi.modules.scalar_cut_plane import ScalarCutPlan
```

and

```
scp = ScalarCutPlane() # set scp as ScalarCutPlane() module
script.add_module(scp) # add module to the scene
scp.implicit_plane.normal = (1, 0, 0) # set normal to Ox axis
# set origin to (i=10, j=25, k=25) i.e. integers for a structured g
scp.implicit_plane.origin = (10, 25, 25)
# set origin to (x=1.0, y=2.5, z=2.5) i.e. reals for unstructured g
# scp.implicit_plane.origin = (1.0, 2.5, 2.5)
scp.implicit_plane.widget.enabled = False
scp.actor.property.diffuse = 0.0 # set some color properties
scp.actor.property.ambient = 1.0 #
scp.actor.property.opacity = 1.0 #
scp.module_manager.scalar_lut_manager.data_range = [0, 1]
```

[](files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_scp.png

Note that if you enable widget, you will be able to translate (move the mouse to the red frame), change the normal (move the mouse to the grey arrow) of the cutplanes in “real-time” :

[]
(files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_scp_widg_en.png

You can also display the cutplanes as “warped surfaces”, just adding a few lines, setting the scale factor and the normals to be computed (smoother surface) or not:

```
scp.enable_warp_scalar = True
scp.compute_normals = True
scp.warp_scalar.filter.scale_factor = 20
```

[]
(files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_scp_warp.png

Of course, you can add as many cutplanes as you want, oblique or not.

Let’s see now a little more complex example : we want opacity to be set to 0.2 for each cutplane, and contours (#10) for the same cutplanes added. Lines above have been changed as below:

Note: I agree, this is not the best way to write such a code. You can obviously write a method to do the same suff. But this is not the purpose here.

```
### cutplane #1, normal to Ox, opacity = 0.2, representation = surf
scp = ScalarCutPlane()
script.add_module(scp)
scp.implicit_plane.normal = (1, 0, 0)
scp.implicit_plane.origin = (25, 25, 25)
```

```
scp.implicit_plane.widget.enabled = False
scp.actor.property.diffuse = 0.0
scp.actor.property.ambient = 1.0
scp.actor.property.opacity = 0.2
scp.module_manager.scalar_lut_manager.data_range = [0, 1]

### cutplane #2, normal to Oy, opacity = 0.2, representation = surf
scp = ScalarCutPlane()
script.add_module(scp)
scp.implicit_plane.normal = (0, 1, 0)
scp.implicit_plane.origin = (25, 25, 25)
scp.implicit_plane.widget.enabled = False
scp.actor.property.diffuse = 0.0
scp.actor.property.ambient = 1.0
scp.actor.property.opacity = 0.2
scp.module_manager.scalar_lut_manager.data_range = [0, 1]

### cutplane #3, normal to Oz, opacity = 0.2, representation = surf
scp = ScalarCutPlane()
script.add_module(scp)
scp.implicit_plane.normal = (0, 0, 1)
scp.implicit_plane.origin = (25, 25, 25)
scp.implicit_plane.widget.enabled = False
scp.actor.property.diffuse = 0.0
scp.actor.property.ambient = 1.0
scp.actor.property.opacity = 0.2
scp.module_manager.scalar_lut_manager.data_range = [0, 1]

### cutplane #4, normal to Ox, representation = contour
scp = ScalarCutPlane()
script.add_module(scp)
scp.implicit_plane.normal = (1, 0, 0)
scp.implicit_plane.origin = (25, 25, 25)
scp.implicit_plane.widget.enabled = False
scp.actor.property.diffuse = 0.0
scp.actor.property.ambient = 1.0
scp.enable_contours = True
scp.contour.number_of_contours = 10
scp.contour.minimum_contour, scp.contour.maximum_contour = [0, 1]
scp.module_manager.scalar_lut_manager.data_range = [0, 1]

### cutplane #5, normal to Oy, representation = contour
scp = ScalarCutPlane()
script.add_module(scp)
scp.implicit_plane.normal = (0, 1, 0)
scp.implicit_plane.origin = (25, 25, 25)
scp.implicit_plane.widget.enabled = False
scp.actor.property.diffuse = 0.0
scp.actor.property.ambient = 1.0
scp.enable_contours = True
scp.contour.number_of_contours = 10
scp.contour.minimum_contour, scp.contour.maximum_contour = [0, 1]
scp.module_manager.scalar_lut_manager.data_range = [0, 1]
```

```
### cutplane #6, normal to Oz, representation = contour
scp = ScalarCutPlane()
script.add_module(scp)
scp.implicit_plane.normal = (0, 0, 1)
scp.implicit_plane.origin = (25, 25, 25)
scp.implicit_plane.widget.enabled = False
scp.actor.property.diffuse = 0.0
scp.actor.property.ambient = 1.0
scp.enable_contours = True
scp.contour.number_of_contours = 10
scp.contour.minimum_contour, scp.contour.maximum_contour = [0, 1]
scp.module_manager.scalar_lut_manager.data_range = [0, 1]
```

which looks like this:

[](files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_scp2.png

Another module that slices grid is called !SliceUnstructuredGrid. As it is called, it should work only for unstructured grids. But, because it has been tested on a structured grid, even !MayaVi2 complains about it with a warning message, it “works” even for structured grid (happily for our example ;-))

In fact, its interest is not really slicing grid, but even more showing the structure of your mesh, i.e. your mesh cells. Thus you can see if there is not any problem (holes, etc.).

```
from enthought.mayavi.modules.slice_unstructured_grid import SliceU
```

and

```
sug = SliceUnstructuredGrid()
script.add_module(sug)
# unstructured grid so origin coordinates are reals
sug.implicit_plane.origin = (25., 25., 25.)
sug.implicit_plane.normal = (1, 1, 1)
sug.implicit_plane.widget.enabled = False
sug.extract_geometry.extract_boundary_cells = False
sug.extract_geometry.extract_inside = True
sug.extract_geometry.extract_only_boundary_cells = False
sug.geom_filter.cell_clipping = False
sug.geom_filter.extent_clipping = False
sug.geom_filter.merging = True
sug.geom_filter.point_clipping = False
sug.actor.property.representation = 'wireframe'
sug.actor.property.diffuse = 0.0
sug.actor.property.ambient = 1.0
sug.actor.property.opacity = 1.0
sug.module_manager.scalar_lut_manager.data_range = [0, 1]
```

The scene should look like this:

[](files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_sug.png

GridPlane/StructuredGridOutline module

Using !GridPlane module cuts also your grid, but quite differently from !ScalarCutPlane module. You can’t get normal plane only along Ox, Oy and Oz axis, and it works only for structured grids. But unlike !ScalarCutPlane module, which always cuts your mesh in a plane, !GridPlane cuts through your mesh: if it’s a conformal mesh, the cut won’t be a plane, but something following the curvature of your mesh.

The !StructuredGridOutline module does the same as Outline module, but for conformal mesh.

To illustrate how can we use these modules, let’s consider a example provided in the VTKData directory, combxyz.bin & combq.bin files (Plot3D format) from the tarball vtkdata-5.0.3.tar.gz you can download here.

So, type:

```
from enthought.mayavi.modules.strucured_grid_outline import Structu
from enthought.mayavi.modules.grid_plane import GridPlane

# to load Plot3D files format
from enthought.mayavi.sources.plot3d_reader import PLOT3DReader
```

and

```
src = PLOT3DReader()
src.initialize('combxyz.bin', 'combq.bin')
script.add_source(src)

sgo = StructuredGridOutline()
script.add_module(sgo)

gp = GridPlane()
script.add_module(gp)
gp.grid_plane.axis = 'x'
gp.grid_plane.position = 2
gp.actor.mapper.scalar_visibility = True
gp.actor.property.representation = 'surface'
gp.actor.property.diffuse = 0.0
gp.actor.property.ambient = 1.0
gp.actor.property.opacity = 1

gp = GridPlane()
script.add_module(gp)
gp.grid_plane.axis = 'x'
gp.grid_plane.position = 25
gp.actor.mapper.scalar_visibility = True
gp.actor.property.representation = 'surface'
gp.actor.property.diffuse = 0.0
gp.actor.property.ambient = 1.0
gp.actor.property.opacity = 1

gp = GridPlane()
script.add_module(gp)
gp.grid_plane.axis = 'x'
gp.grid_plane.position = 55
gp.actor.mapper.scalar_visibility = True
gp.actor.property.representation = 'surface'
gp.actor.property.diffuse = 0.0
gp.actor.property.ambient = 1.0
gp.actor.property.opacity = 1
```

The scene is rendered as this:

[](files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_sgo_gp.png

Surface/IsoSurface module

Others modules are Surface and !IsoSurface. These modules work with any data.

Surface module does the same as !IsoSurface but displays, automatically, several isosurfaces for a given number of values in a given range.

In fact, you can get the same result with !IsoSurface module, but you will have to set each isovalue.

When several isosurfaces are displayed, using Surface or !IsoSurface module, you should set opacity to a value below 1, in order to see all isosurfaces.

Using Surface module is straightforward:

```
from enthought.mayavi.modules.surface import Surface
```

then

```
s = Surface()
s.enable_contours = True # we want contours enabled
s.contour.auto_contours = True # we want isovalues automatically we
s.contour.number_of_contours = 10 # self-explanatory ;-)
s.actor.property.opacity = 0.2
script.add_module(s)
s.contour.minimum_contour = 0
s.contour.maximum_contour = 1
s.module_manager.scalar_lut_manager.data_range = [0, 1]
```

The scene should look like this:

[](files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_surface.png

Using the !IsoSurface module is not more difficult. As an example, say that we want the same result as the Surface module displays.

```
from enthought.mayavi.modules.iso_surface import IsoSurface
```

and

```
isosurf = IsoSurface()
script.add_module(isosurf)
isosurf.contour.contours = [0.1111, 0.2222, 0.3333, 0.4444, 0.5555,
isosurf.compute_normals = True
isosurf.actor.property.opacity = 0.2
isosurf.module_manager.scalar_lut_manager.data_range = [0, 1]
```

This yelds the same scene as previous, of course, but now, you can control each isovalue separately.

[]
(files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_isosurface.pn
g

The funny part is that you can set the minimum/maximum contour for Surface or Contours for !IsoSurface in “real-time”, moving the slide-bar. This is a very useful feature. And can render very nice “dynamic” scene ! :-)

Volume module

It is still quite experimental for me (you can set a lot of parameters), so this section will be very short ;-)

Instead of viewing surfaces, data are displayed in the whole volume.

Begin to import the required module:

```
from enthought.mayavi.modules.volume import Volume
```

and then, add it to the source as usual:

```
v = Volume()
script.add_module(v)
v.lut_manager.show_scalar_bar = True
v.lut_manager.scalar_bar.orientation = 'vertical'
v.lut_manager.scalar_bar.width = 0.1
v.lut_manager.scalar_bar.height = 0.8
v.lut_manager.scalar_bar.position = (0.01, 0.15)
v.lut_manager.scalar_bar.label_text_property.color = fg_color
v.lut_manager.scalar_bar.title_text_property.color = fg_color
v.lut_manager.number_of_labels = 10
v.lut_manager.data_name = ""
```

Note that the Volume module has a “Color Transfer Function”, which is quite different from the !LookUp Table used by the others modules.

The rendered scene should look like this (thanks to Prabhu to have made the CTF similar to the LUT) :

[](files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_volume.png

Vectors/Glyph/VectorCutPlane/WarpVectorCutPlan e module

Until now, we have only dealt with scalar values. You can also display values as vectors. You can use one of the three following modules:

```
* Vectors module: scale and color are set by vectors data, i.e. a 3I
```

```
* Glyph module: scale and color are set by scalar data;
```

```
* !VectorCutPlane module; in this case, vectors are not diplayed in
```

You can set several parameters for these modules, in concern with arrows shape, etc.

First, it depends of the number of points in your volume, but you are advised to decimate your data. If you don't, you should see nothing all but a lot of arrows everywhere, and thus loss the pertinent information. You can choose a randomly, or not, decimation.

Second, you can choose the shape of your vectors, amongst the following list: 2D Glyph or Arrow, Cone, Cylinder, Sphere and Cube 3D vector shapes.

Third, you can set some parameters for the choosen shape. For example, using the Arrow shape, you can set the following properties for the shaft and the tip:

```
* the shaft radius;
```

```
* the shaft resolution (number of polygons);
```

```
* the tip length;
```

```
* the tip radius;
```

```
* the tip resolution;
```

You can also set the vector position, between "tail", "centered" and "head", the scale mode, the color mode, the scale factor (how big your vectors will be displayed), etc.

Let's see now how one can do this.

First, import the required module.

For Vectors module,

```
from enthought.mayavi.modules.vectors import Vectors
```

For Glyph module,

```
from enthought.mayavi.modules.glyph import Glyph
```

For !VectorCutPlane module,

```
from enthought.mayavi.modules.vector_cut_plane import VectorCutPlar
```

In fact, you will see that these three modules use the same objects and methods. Only default values differ.

For instance, for Vectors module, you can type:

```
v = Vectors()
script.add_module(v)
v.glyph.mask_input_points = True             # we want to decimate
v.glyph.mask_points.on_ratio = 100           # ...by a ratio of 10(
v.glyph.mask_points.random_mode = True       # I want a randomly de
v.glyph.glyph_source = v.glyph.glyph_list[1] # I like ArrowSource ;
# following values are the default values: tweak your own !
v.glyph.glyph_source.shaft_radius = 0.03
v.glyph.glyph_source.shaft_resolution = 6
v.glyph.glyph_source.tip_length = 0.35
v.glyph.glyph_source.tip_radius = 0.1
v.glyph.glyph_source.tip_resolution = 6
v.glyph.glyph.scale_factor = 10
v.glyph.glyph_position = 'tail'
v.glyph.scale_mode = 'scale_by_vector'
v.glyph.color_mode = 'color_by_vector'
### if you use Glyph module, here are the default values
# v.glyph.glyph_position = 'center'
# v.glyph.scale_mode = 'scale_by_scalar'
# v.glyph.color_mode = 'color_by_scalar'
```

If we consider, once again ;-), the same 3D data already shown before, but this time, with vectors instead of scalars data, the scene should look like this:

[](files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_vectors.png

For the !VectorCutPlane module, you can set the same properties as above plus the properties of the !ScalarCutPlane module such as implicit_plane.normal, implicit_plane.origin, implicit_plane.widget.enabled, etc:

```
vcp = VectorCutPlane()
script.add_module(vcp)
vcp.glyph.mask_input_points = True
vcp.glyph.mask_points.on_ratio = 5
vcp.glyph.mask_points.random_mode = False
vcp.glyph.glyph_source = vcp.glyph.glyph_list[1]
vcp.glyph.glyph_source.shaft_radius = 0.03
vcp.glyph.glyph_source.shaft_resolution = 6
vcp.glyph.glyph_source.tip_length = 0.35
vcp.glyph.glyph_source.tip_radius = 0.1
vcp.glyph.glyph_source.tip_resolution = 6
vcp.glyph.glyph.scale_factor = 20
vcp.glyph.glyph_position = 'tail'
vcp.glyph.scale_mode = 'scale_by_vector'
vcp.glyph.color_mode = 'color_by_vector'
vcp.implicit_plane.normal = (1, 0, 0) # set normal to Ox axis
vcp.implicit_plane.origin = (10, 25, 25) # set origin to (i=10, j=2
vcp.implicit_plane.widget.enabled = True
vcp.actor.property.diffuse = 0.0 # set some color properties
vcp.actor.property.ambient = 1.0 #
vcp.actor.property.opacity = 1.0 #
vcp.module_manager.vector_lut_manager.data_range = [0, 1]
```

This should render this scene:

[](files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_vcp.png

You can also warp a cutplane according to the vectors field. To do this, you have to load another module, instead of !VectorCutPlane, called !WarpVectorCutPlane.

Type:

```
from enthought.mayavi.modules.warp_vector_cut_plane import WarpVect
```

then

```
wvcp = WarpVectorCutPlane()
script.add_module(wvcp)
wvcp.implicit_plane.normal = (1, 0, 0) # set normal to Ox axis
wvcp.implicit_plane.origin = (10, 25, 25) # set origin to (i=10, j=
wvcp.implicit_plane.widget.enabled = True
wvcp.compute_normals = True
wvcp.warp_vector.filter.scale_factor = 10
```

You should get this (compare to the warped surface with !ScalarCutPlane module):

[]
(files/attachments/MayaVi_ScriptingMayavi2_MainModules/module_warpvcp.png

Streamline module

Another way to display vectors fields is to use the Streamline module.

We consider here others Plot3D files: postxyz.bin & postq.bin that you can download here. You can find some screenshots using these files on the VTK home page here.

You can set several parameters for this module: for instance, the type of the streamline (tube, ribbon or line) with its properties, and the “seed”.

We also use the !GridPlane module in this example:

Begin to import the required module:

```
from enthought.mayavi.sources.plot3d_reader import PLOT3DReader
from enthought.mayavi.modules.streamline import Streamline
from enthought.mayavi.modules.grid_plane import GridPlane
```

In this example, we want streamlines displayed as tubes, with 10 sides, and the seed set to the line seed. We also choose to display the “Kinetic Energy” part of the Plot3D files.

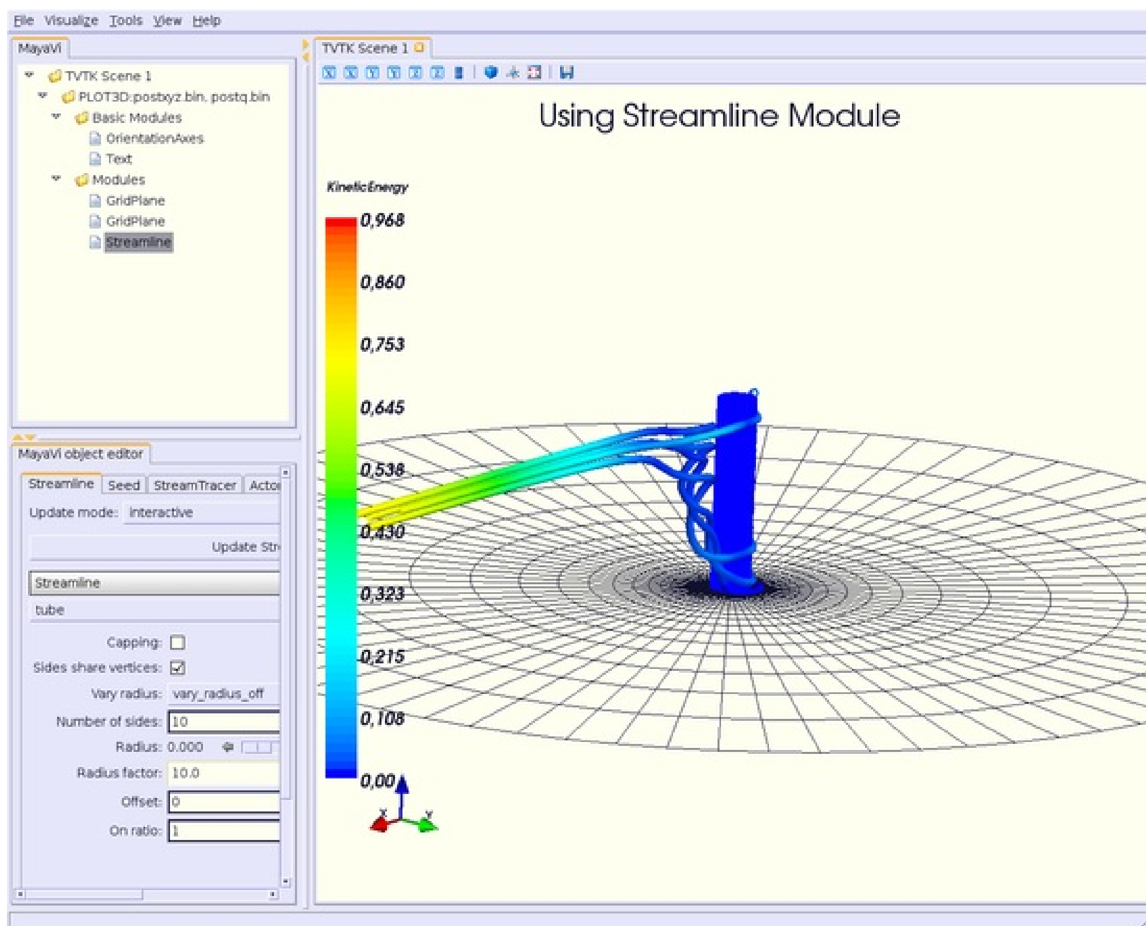
```
src = PLOT3DReader()
src.initialize('postxyz.bin', 'postq.bin')
src.scalars_name = "kinetic energy"
script.add_source(src)

gp = GridPlane()
script.add_module(gp)
gp.grid_plane.axis = 'x'
gp.actor.mapper.scalar_visibility = True
gp.actor.property.representation = 'surface'
gp.actor.property.diffuse = 0.0
gp.actor.property.ambient = 1.0
gp.actor.property.opacity = 1

gp = GridPlane()
script.add_module(gp)
gp.grid_plane.axis = 'z'
gp.actor.mapper.scalar_visibility = False
gp.actor.property.representation = 'wireframe'
gp.actor.property.diffuse = 0.0
gp.actor.property.ambient = 1.0
gp.actor.property.opacity = 1

strl = Streamline()
script.add_module(strl)
strl.streamline_type = "tube" # tube, ribbon or line
strl.tube_filter.number_of_sides = 10
strl.tube_filter.radius = 0.1
strl.seed.widget = strl.seed.widget_list[1] # [Sphere, Line, Plane,
strl.seed.widget.align = "z_axis" # or "x_axis", "y_axis"
strl.seed.widget.point1 = (-0.7, 0, 0)
strl.seed.widget.point2 = (-0.7, 0, 4.82)
strl.seed.widget.resolution = 10
strl.seed.widget.enabled = False
```

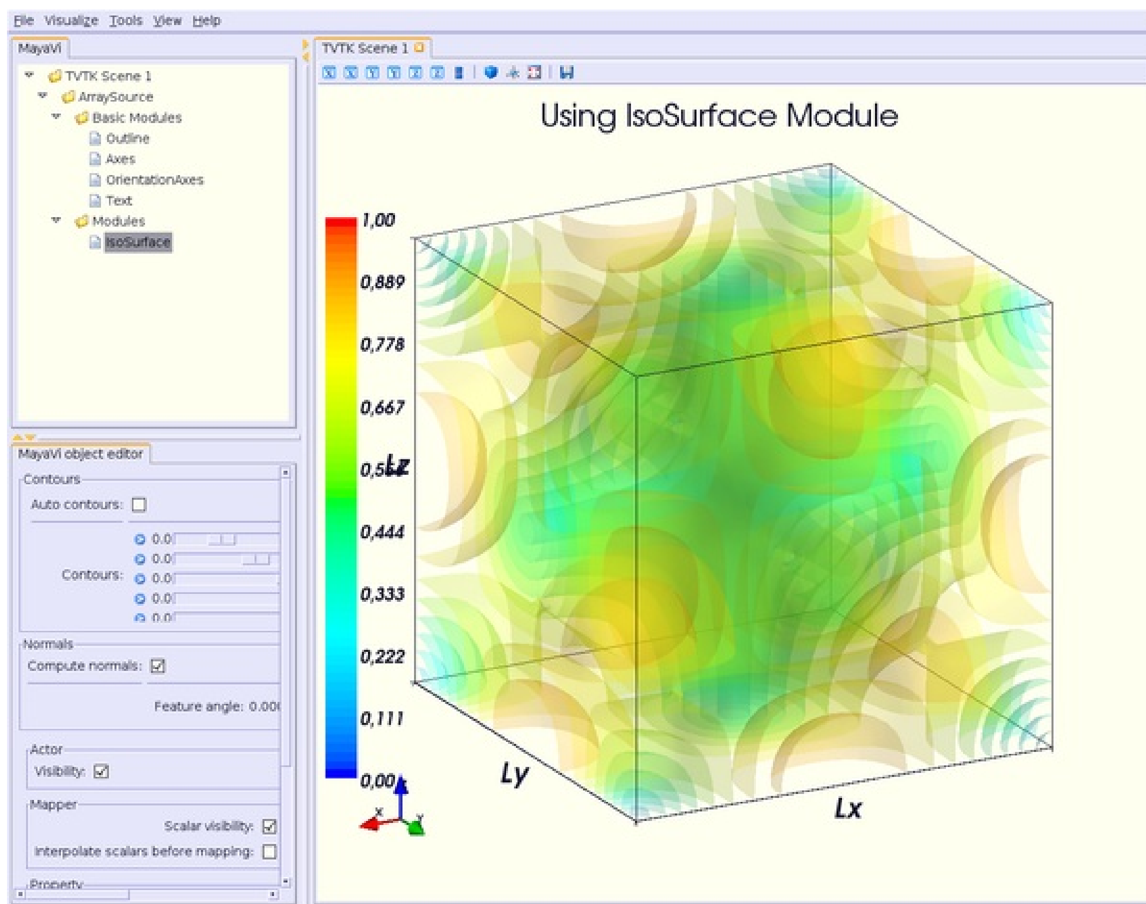
This should look like:

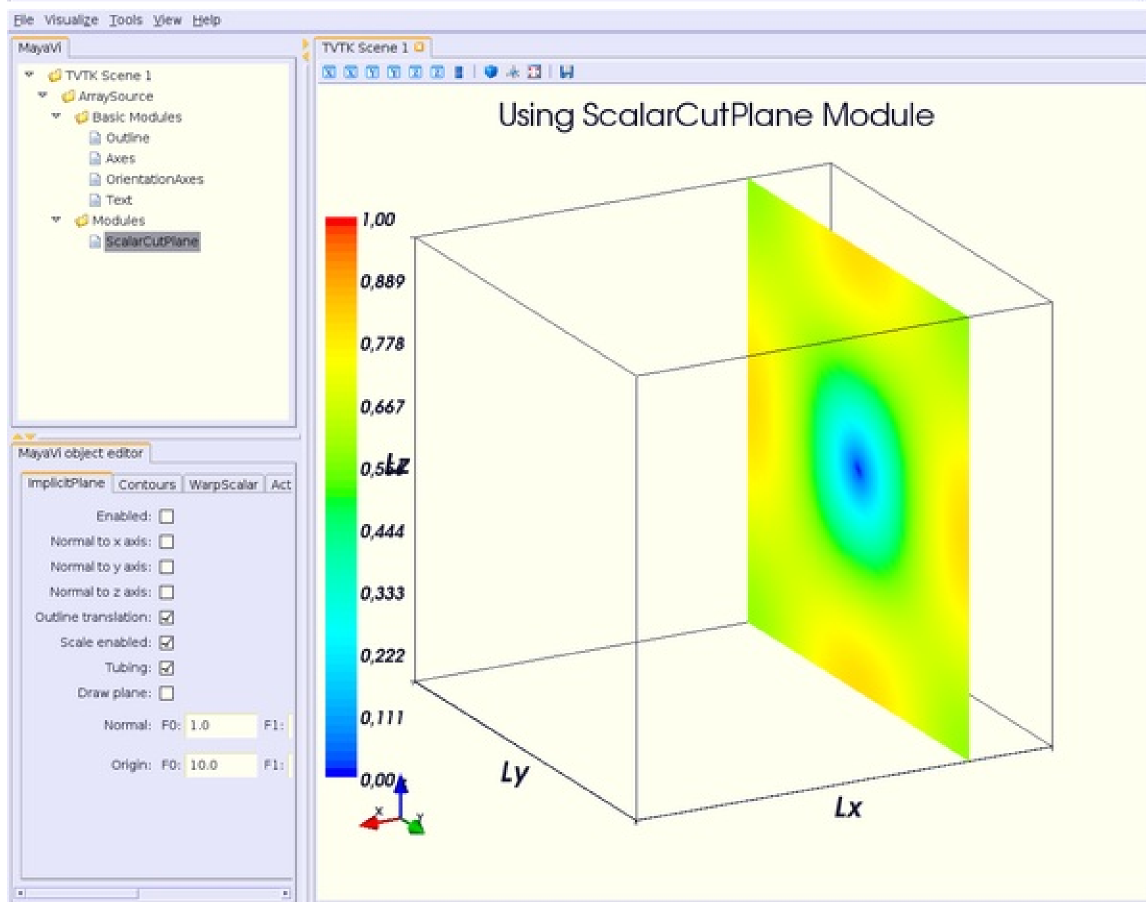


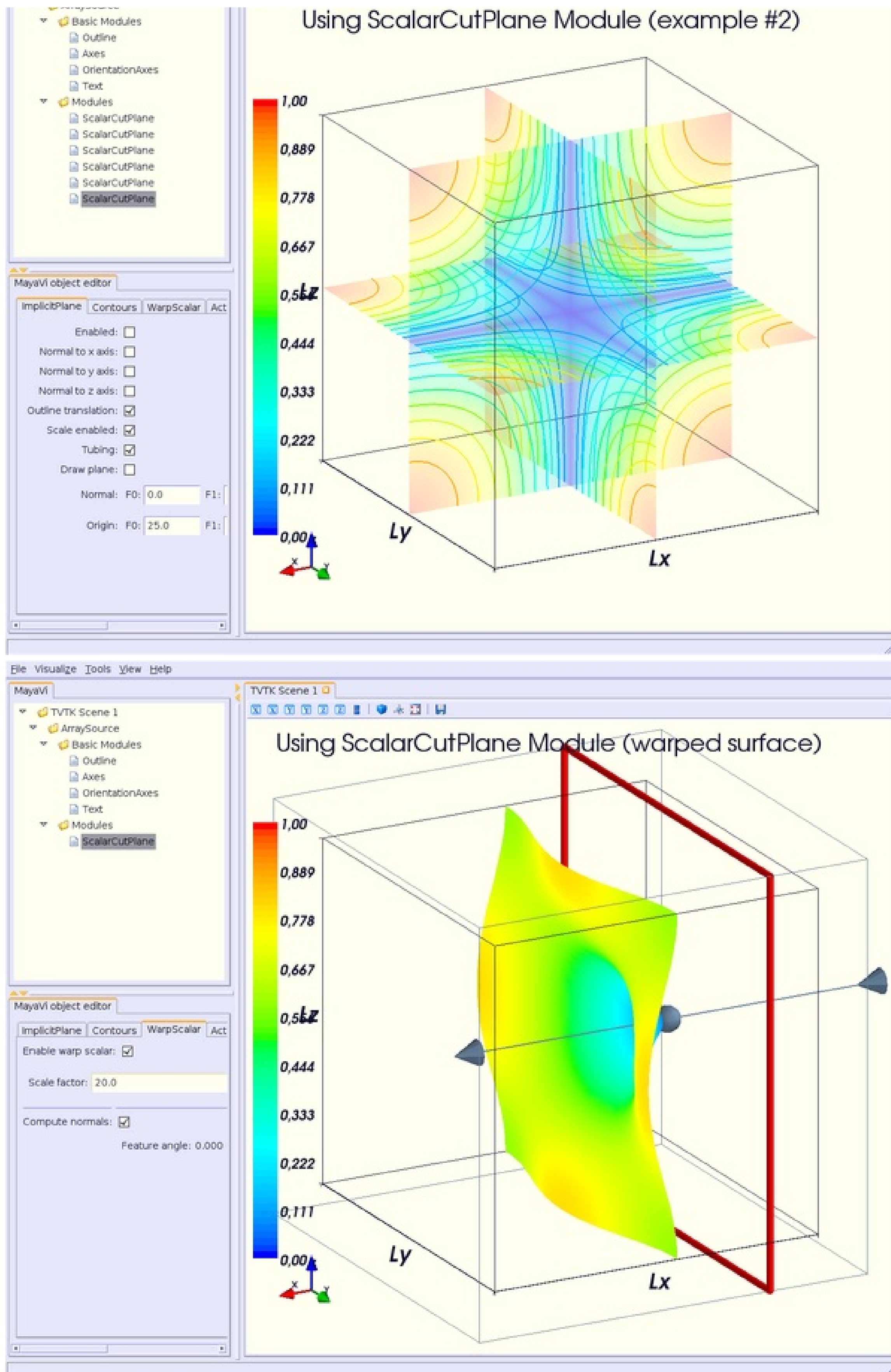
Note: you can also see an example of using the Streamline module in [:Cookbook/MayaVi/Examples: Cookbook/MayaVi/Examples].

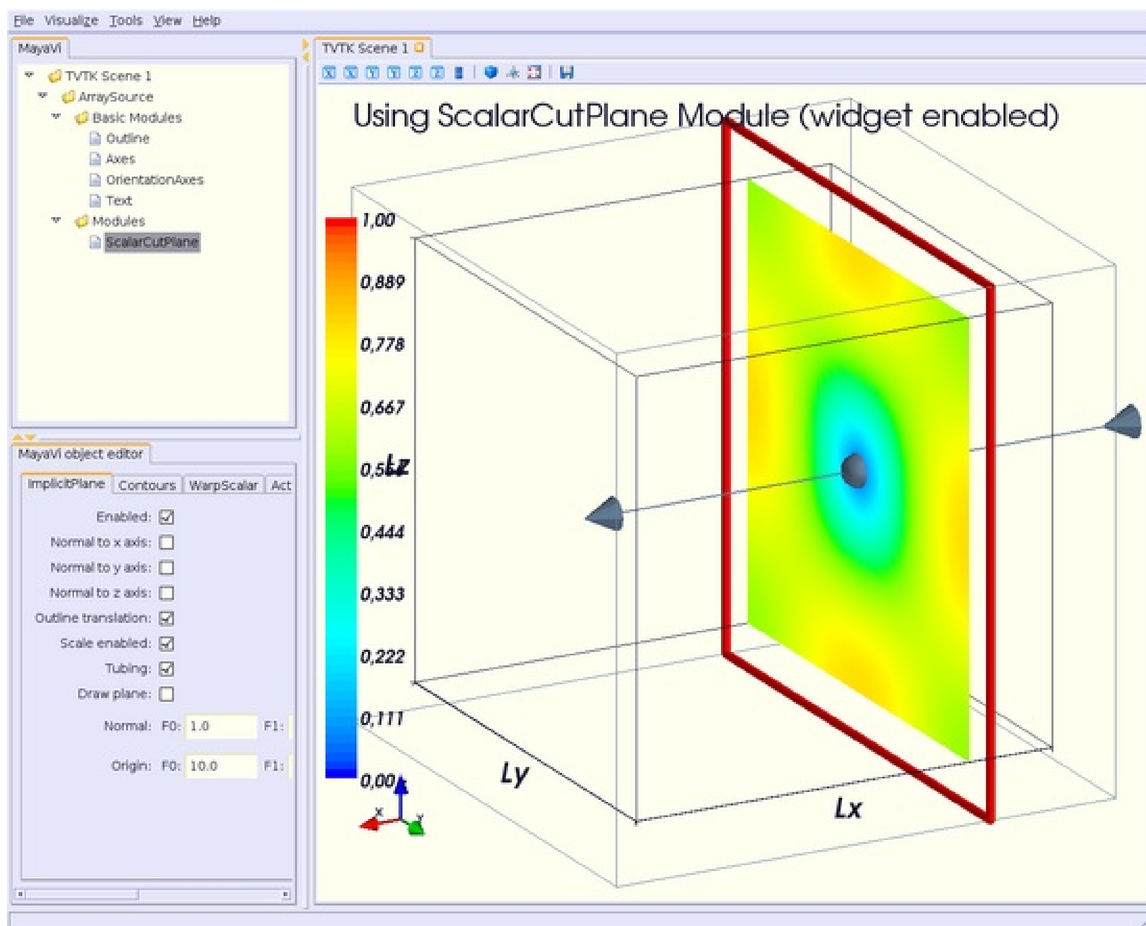
Attachments

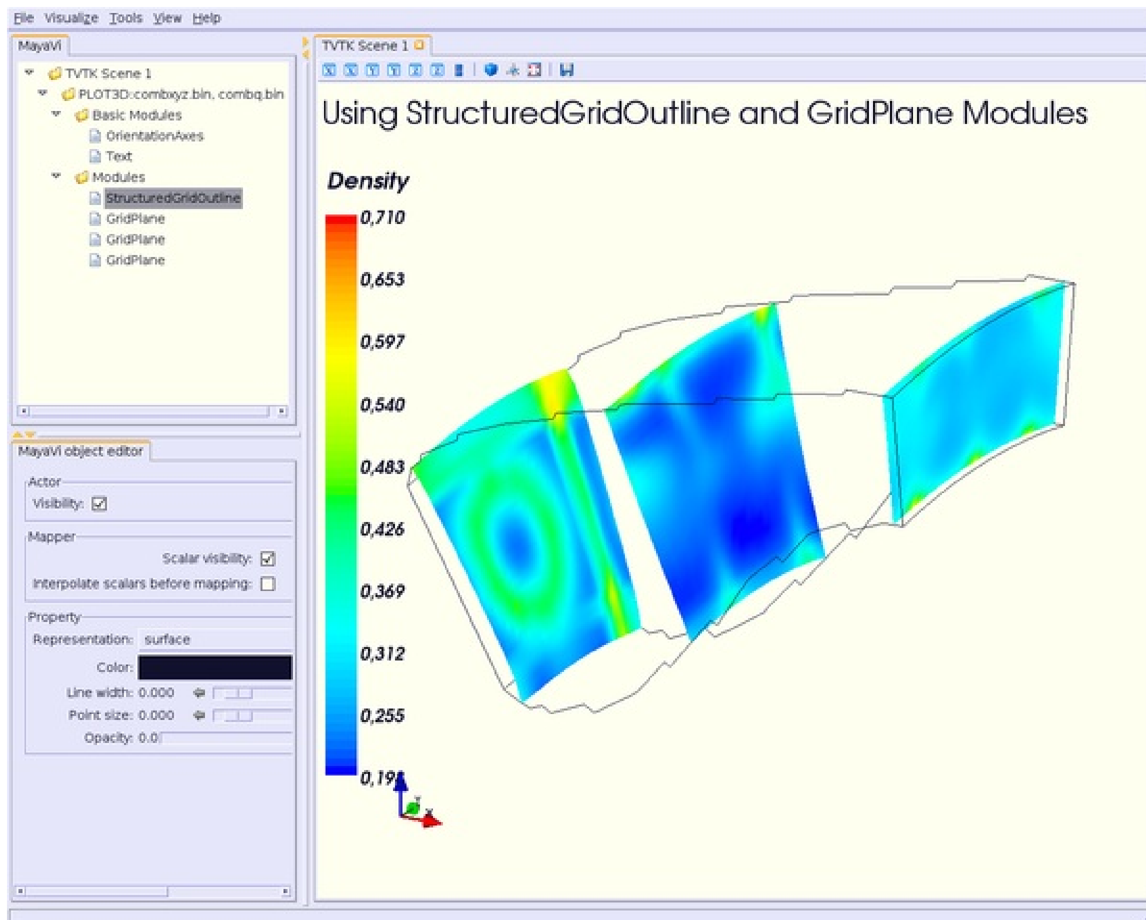
- `module_isosurface.png`
- `module_scp.png`
- `module_scp2.png`
- `module_scp_warp.png`
- `module_scp_widg_en.png`
- `module_sgo_gp.png`
- `module_streamline.png`
- `module_sug.png`
- `module_surface.png`
- `module_vcp.png`
- `module_vectors.png`
- `module_volume.png`
- `module_warpvcp.png`

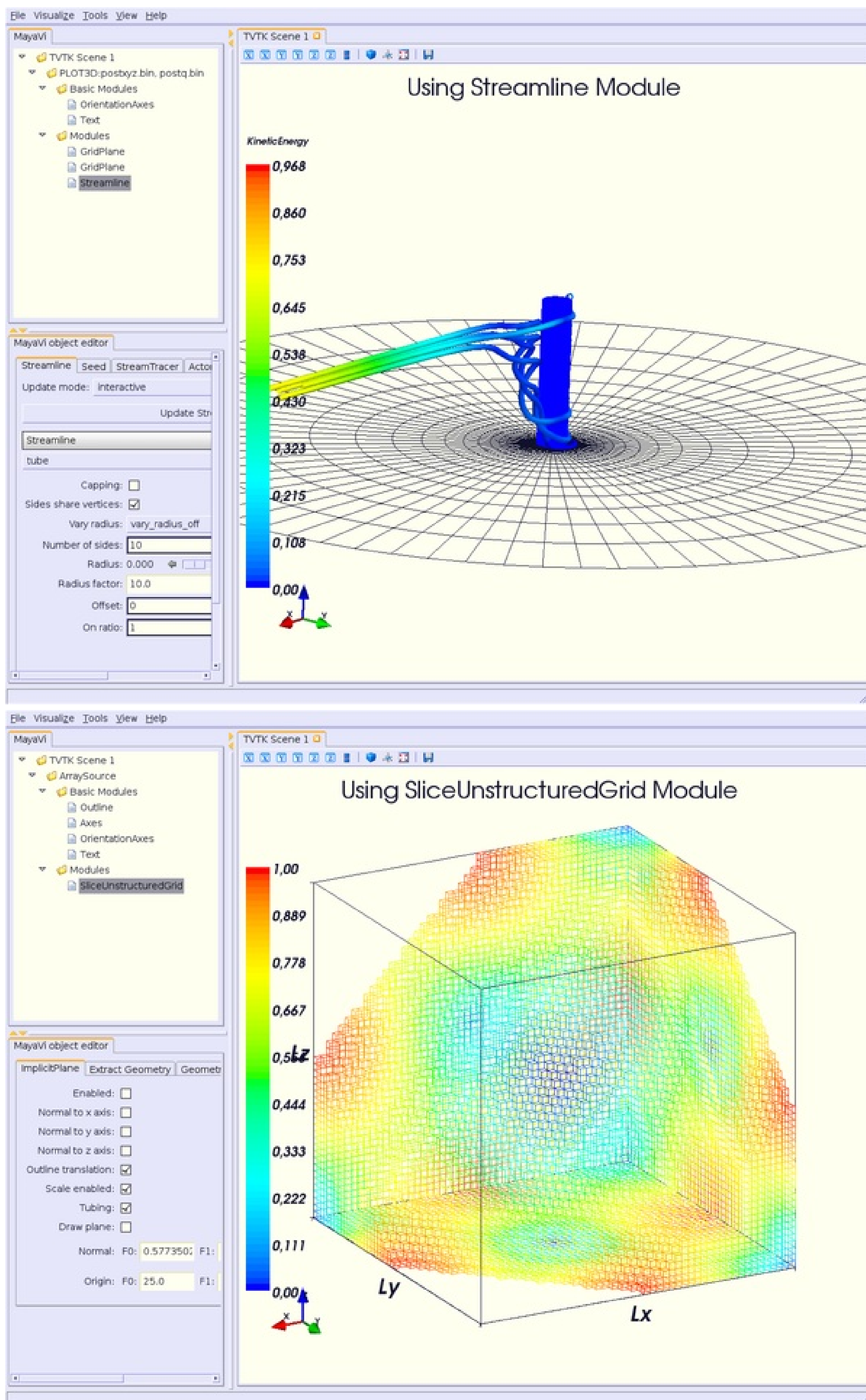

File Visualize Tools View Help
MayaVi
TVTK Scene 1
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
Modules
IsoSurface
MayaVi object editor
Contours
Auto contours:
Contours:
Normals
Compute normals:
Feature angle:
Actor
Visibility:
Mapper
Scalar visibility:
Interpolate scalars before mapping:
Using IsoSurface Module
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111
0,00
Lz
Ly
Lx


File Visualize Tools View Help
MayaVi
TVTK Scene 1
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
Modules
ScalarCutPlane
MayaVi object editor
ImplicitPlane
Contours
WarpScalar
Act
Enabled:
Normal to x axis:
Normal to y axis:
Normal to z axis:
Outline translation:
Scale enabled:
Tubing:
Draw plane:
Normal: F0: 1.0 F1:
Origin: F0: 10.0 F1:
Using ScalarCutPlane Module
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111
0,00
Lz
Ly
Lx

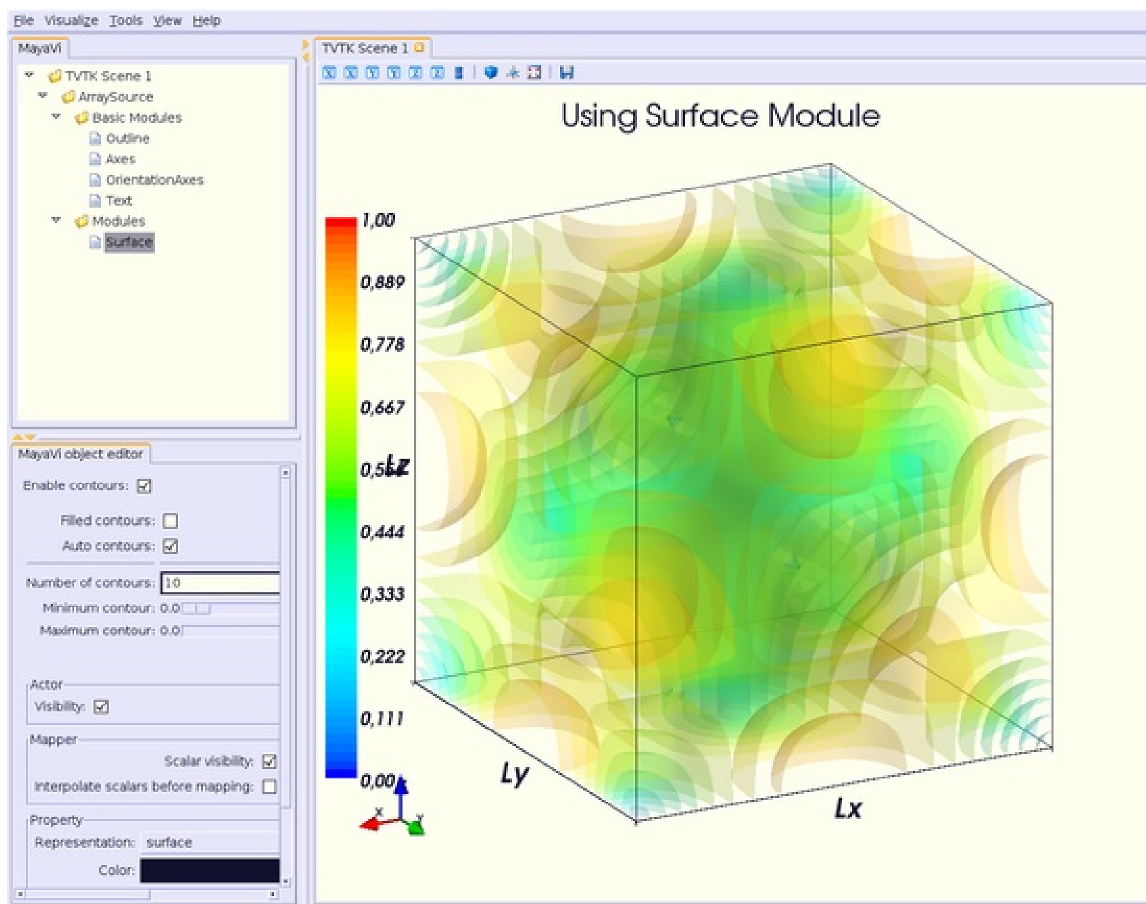

Using ScalarCutPlane Module (example #2)
Basic Modules
Outline
Axes
OrientationAxes
Text
Modules
ScalarCutPlane
MayaVi object editor
ImplicitPlane
Contours
WarpScalar
Enabled:
Normal to x axis:
Normal to y axis:
Normal to z axis:
Outline translation:
Scale enabled:
Tubing:
Draw plane:
Normal: F0: 0.0
Origin: F0: 25.0
Ly
Lx
File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Using ScalarCutPlane Module (warped surface)
Enable warp scalar:
Scale factor: 20.0
Compute normals:
Feature angle: 0.000
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111

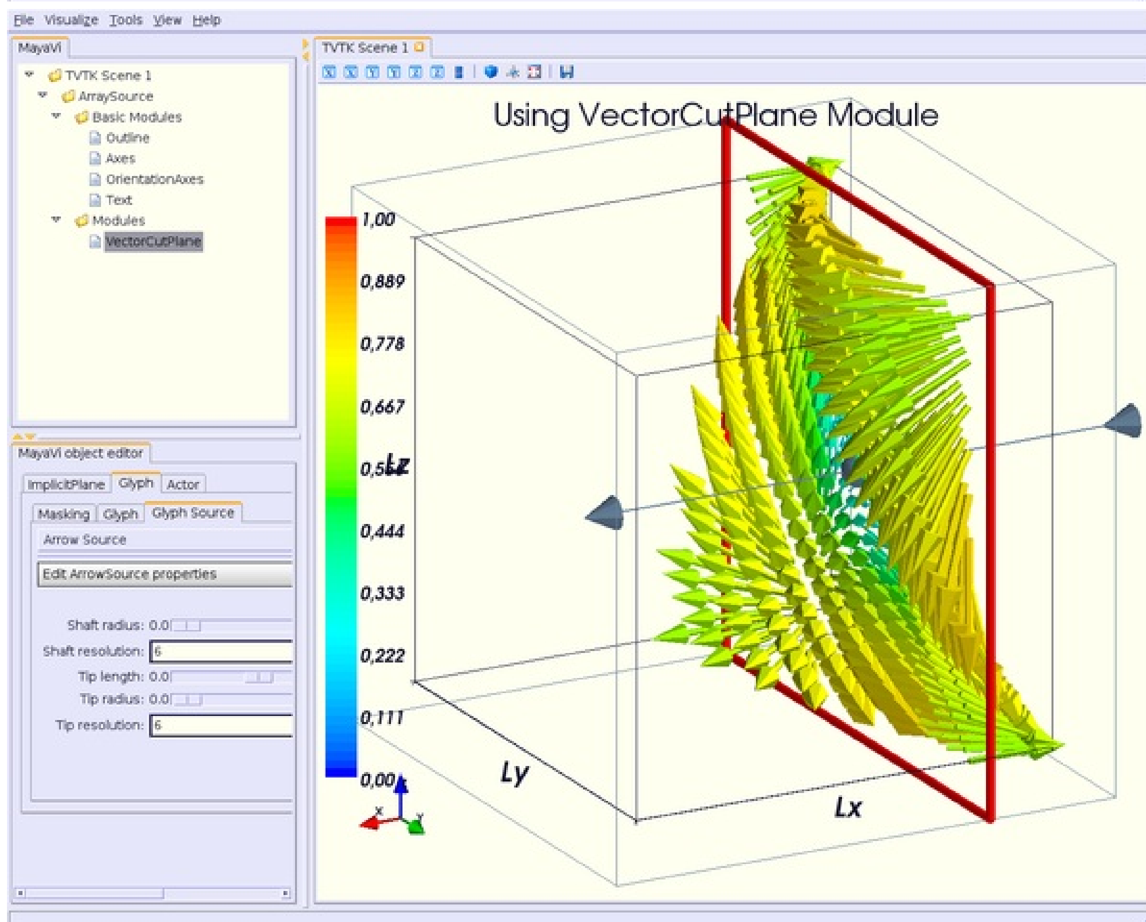

File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
Modules
ScalarCutPlane
MayaVi object editor
ImplicitPlane
Contours
WarpScalar
Act
Enabled:
Normal to x axis:
Normal to y axis:
Normal to z axis:
Outline translation:
Scale enabled:
Tubing:
Draw plane:
Normal: F0: 1.0 F1:
Origin: F0: 10.0 F1:
Using ScalarCutPlane Module (widget enabled)
1,00
0,889
0,778
0,667
Lz
0,444
0,333
0,222
0,111
0,00
Ly
Lx

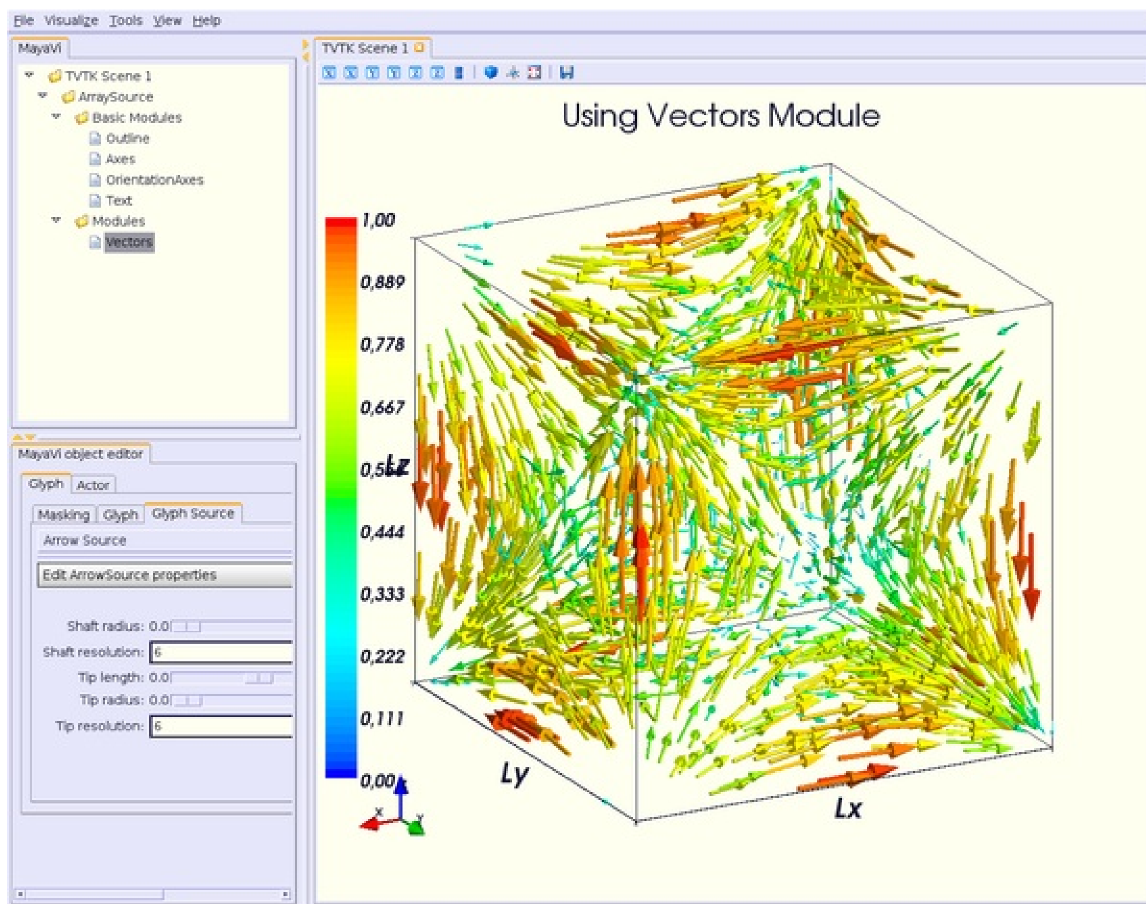

File Visualize Tools View Help
MayaVi
TVTK Scene 1
PLOT3D:combxyz.bin, combq.bin
Basic Modules
OrientationAxes
Text
Modules
StructuredGridOutline
GridPlane
GridPlane
GridPlane
MayaVi object editor
Actor
Visibility:
Mapper
Scalar visibility:
Interpolate scalars before mapping:
Property
Representation: surface
Color:
Line width: 0.000
Point size: 0.000
Opacity: 0.0
Using StructuredGridOutline and GridPlane Modules
Density
0,710
0,653
0,597
0,540
0,483
0,426
0,369
0,312
0,255

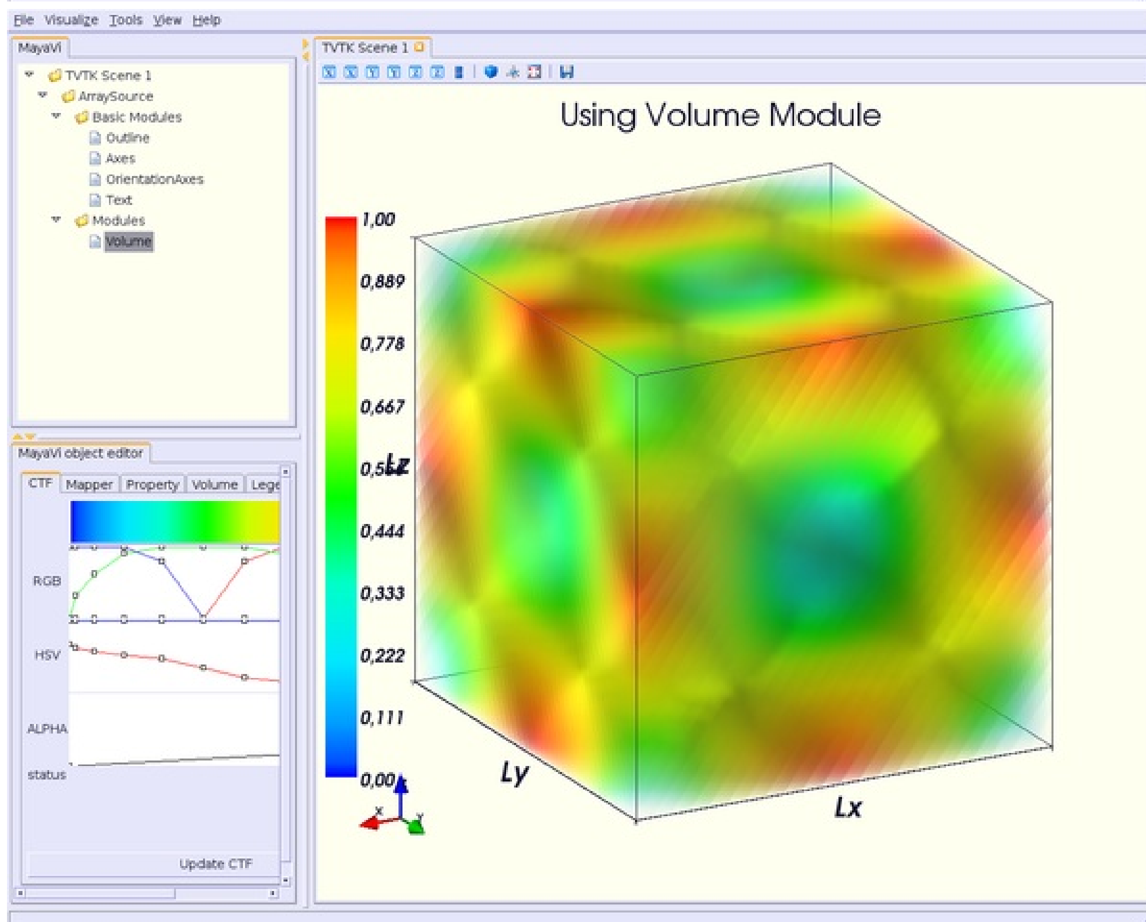


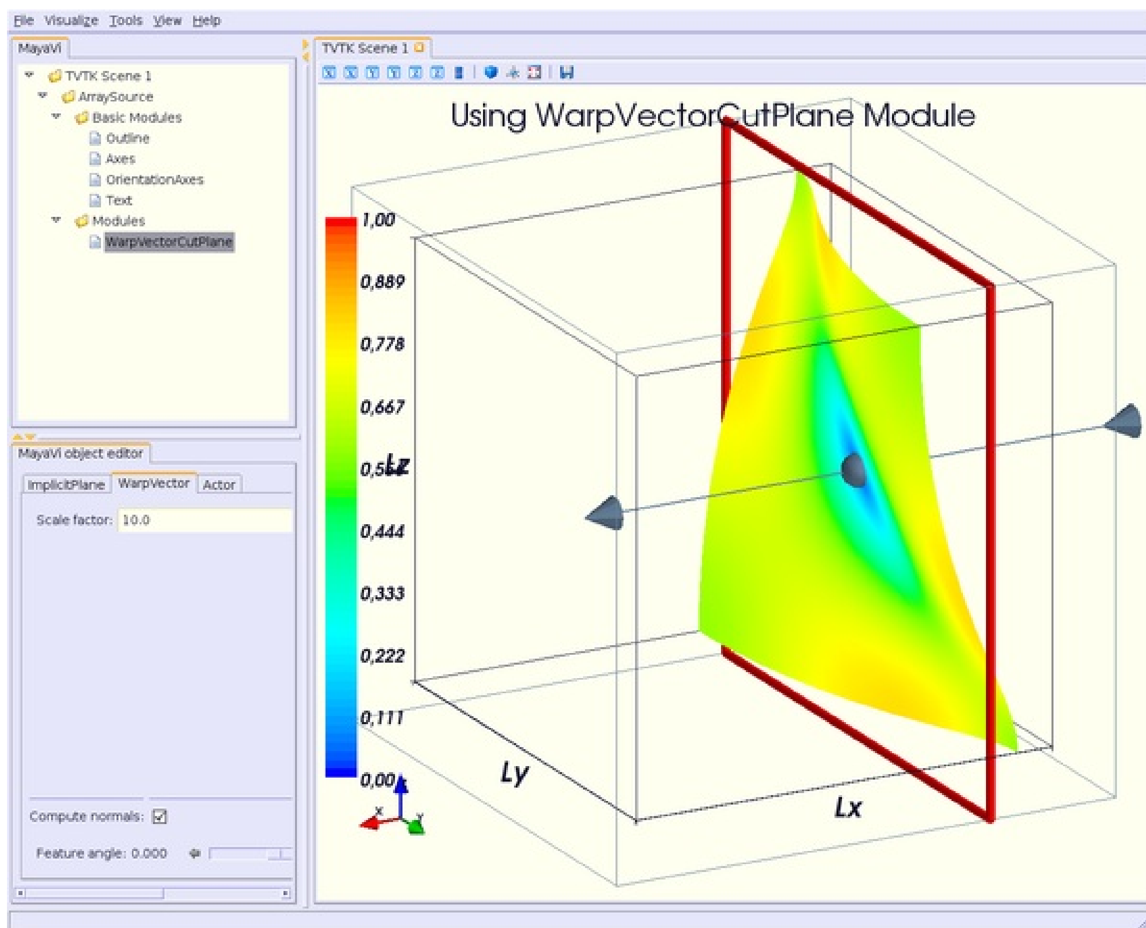
File Visualize Tools View Help
MayaVi
TVTK Scene 1
PLOT3D:postxyz.bin, postq.bin
Basic Modules
OrientationAxes
Text
Modules
GridPlane
GridPlane
Streamline
MayaVi object editor
Streamline
Seed
StreamTracer
Update mode:
interactive
Streamline
tube
Capping:
Sides share vertices:
Vary radius:
vary_radius_off
Number of sides:
10
Radius:
0.000
Radius factor:
10.0
Offset:
0
On ratio:
1
Using Streamline Module
KineticEnergy
0,968
0,860
0,753
0,645
0,538
0,430
0,323
0,215
0,108
0,00
File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
Modules
SliceUnstructuredGrid
MayaVi object editor
ImplicitPlane
Extract Geometry
Enabled:
Normal to x axis:
Normal to y axis:
Normal to z axis:
Outline translation:
Scale enabled:
Tubing:
Draw plane:
Normal:
F0:
F1:
Origin:
F0:
25.0
F1:
Using SliceUnstructuredGrid Module
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111
0,00
Lz
Ly
Lx


File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
Modules
Surface
MayaVi object editor
Enable contours:
Filled contours:
Auto contours:
Number of contours: 10
Minimum contour: 0.0
Maximum contour: 0.0
Actor
Visibility:
Mapper
Scalar visibility:
Interpolate scalars before mapping:
Property
Representation: surface
Color:
Using Surface Module
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111
Ly
Lx
Lz


File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
Modules
VectorCutPlane
MayaVi object editor
ImplicitPlane
Glyph
Actor
Masking
Glyph Source
Arrow Source
Edit ArrowSource properties
Shaft radius: 0.0
Shaft resolution: 6
Tip length: 0.0
Tip radius: 0.0
Tip resolution: 6
Using VectorCutPlane Module
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111
Ly
Lx
Lz


File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
Modules
Vectors
MayaVi object editor
Glyph
Actor
Masking
Glyph
Glyph Source
Arrow Source
Edit ArrowSource properties
Shaft radius: 0.0
Shaft resolution: 6
Tip length: 0.0
Tip radius: 0.0
Tip resolution: 6
Using Vectors Module
1,00
0,889
0,778
0,667
Lz
0,444
0,333
0,222
0,111
0,00
Ly
Lx


File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
Modules
Volume
MayaVi object editor
CTF
Mapper
Property
Volume
RGB
HSV
ALPHA
status
Update CTF
Using Volume Module
1,00
0,889
0,778
0,667
Lz
0,444
0,333
0,222
0,111
0,00
Ly
Lx


File Visualize Tools View Help
MayaVi
TVTK Scene 1
ArraySource
Basic Modules
Outline
Axes
OrientationAxes
Text
Modules
WarpVectorCutPlane
MayaVi object editor
ImplicitPlane
WarpVector
Actor
Scale factor: 10.0
Compute normals:
Feature angle: 0.000
TVTK Scene 1
Using WarpVectorCutPlane Module
1,00
0,889
0,778
0,667
0,444
0,333
0,222
0,111
Lz
Ly
Lx

Performance

A beginners guide to using Python for performance computing

A comparison of weave with NumPy, Pyrex, Psyco, Fortran (77 and 90) and C++ for solving Laplace's equation. This article was originally written by Prabhu Ramachandran.

`laplace.py` is the complete Python code discussed below. The source tarball (`perfpy_2.tgz`) contains in addition the Fortran code, the pure C++ code, the Pyrex sources and a setup.py script to build the f2py and Pyrex module.

Introduction

This is a simple introductory document to using Python for performance computing. We'll use NumPy, SciPy's weave (using both weave.blitz and weave.inline) and Pyrex. We will also show how to use f2py to wrap a Fortran subroutine and call it from within Python.

We will also use this opportunity to benchmark the various ways to solve a particular numerical problem in Python and compare them to an implementation of the algorithm in C++.

Problem description

The example we will consider is a very simple (read, trivial) case of solving the 2D Laplace equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). The formal specification of the problem is as follows. We are required to solve for some unknown function u(x,y) such that ∇2u = 0 with a boundary condition specified. For convenience the domain of interest is considered to be a rectangle and the boundary values at the sides of this rectangle are given.

It can be shown that this problem can be solved using a simple four point averaging scheme as follows. Discretise the domain into an (nx x ny) grid of points. Then the function u can be represented as a 2 dimensional array - u(nx, ny). The values of u along the sides of the rectangle are given. The solution can be obtained by iterating in the following manner.

```
for i in range(1, nx-1):
    for j in range(1, ny-1):
        u[i,j] = ((u[i-1, j] + u[i+1, j])*dy**2 +
                  (u[i, j-1] + u[i, j+1])*dx**2)/(2.0*(dx**2 + dy**
```

Where dx and dy are the lengths along the x and y axis of the discretised domain.

Numerical Solution

Implementing a solver for this is straight forward in Pure Python. Use a simple NumPy array to store the solution matrix u. The following code demonstrates a simple solver.

```
import numpy

class Grid:
    """A simple grid class that stores the details and solution of
    computational grid."""
    def __init__(self, nx=10, ny=10, xmin=0.0, xmax=1.0,
                 ymin=0.0, ymax=1.0):
        self.xmin, self.xmax, self.ymin, self.ymax = xmin, xmax, yr
        self.dx = float(xmax-xmin)/(nx-1)
        self.dy = float(ymax-ymin)/(ny-1)
        self.u = numpy.zeros((nx, ny), 'd')
        # used to compute the change in solution in some of the met
        self.old_u = self.u.copy()

    def setBCFunc(self, func):
        """Sets the BC given a function of two variables."""
        xmin, ymin = self.xmin, self.ymin
        xmax, ymax = self.xmax, self.ymax
        x = numpy.arange(xmin, xmax + self.dx*0.5, self.dx)
        y = numpy.arange(ymin, ymax + self.dy*0.5, self.dy)
        self.u[0 ,:] = func(xmin,y)
        self.u[-1,:] = func(xmax,y)
        self.u[:, 0] = func(x,ymin)
        self.u[:,-1] = func(x,ymax)

    def computeError(self):
        """Computes absolute error using an L2 norm for the solutic
        This requires that self.u and self.old_u must be appropriat
        setup."""
        v = (self.u - self.old_u).flat
        return numpy.sqrt(numpy.dot(v,v))

class LaplaceSolver:
    """A simple Laplacian solver that can use different schemes to
    solve the problem."""
    def __init__(self, grid, stepper='numeric'):
        self.grid = grid
        self.setTimeStepper(stepper)

    def slowTimeStep(self, dt=0.0):
        """Takes a time step using straight forward Python loops."'
        g = self.grid
```

```
        nx, ny = g.u.shape
        dx2, dy2 = g.dx**2, g.dy**2
        dnr_inv = 0.5/(dx2 + dy2)
        u = g.u

        err = 0.0
        for i in range(1, nx-1):
            for j in range(1, ny-1):
                tmp = u[i,j]
                u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
                         (u[i, j-1] + u[i, j+1])*dx2)*dnr_inv
                diff = u[i,j] - tmp
                err += diff*diff

        return numpy.sqrt(err)

    def setTimeStepper(self, stepper='numeric'):
        """Sets the time step scheme to be used while solving giver
        string which should be one of ['slow', 'numeric', 'blitz',
        'inline', 'fastinline', 'fortran']."""
         if stepper == 'slow':
           self.timeStep = self.slowTimeStep
         # ...
         else:
            self.timeStep = self.numericTimeStep

    def solve(self, n_iter=0, eps=1.0e-16):
        err = self.timeStep()
        count = 1

        while err > eps:
            if n_iter and count >= n_iter:
                return err
            err = self.timeStep()
            count = count + 1

        return count
```

The code is pretty simple and very easy to write but if we run it for any sizeable problem (say a 500 x 500 grid of points), we'll see that it takes *forever* to run. The CPU hog in this case is the `slowTimeStep` method. In the next section we will speed it up using NumPy.

Using NumPy

It turns out that the innermost loop of the `LaplaceSolver.slowTimeStep` method can be readily expressed by a much simpler NumPy expression. Here is a re-written timeStep method.

```
def numericTimeStep(self, dt=0.0):
    """Takes a time step using a NumPy expression."""
    g = self.grid
    dx2, dy2 = g.dx**2, g.dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    u = g.u
    g.old_u = u.copy() # needed to compute the error.

    # The actual iteration
    u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                     (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv

    return g.computeError()
```

The entire for i and j loops have been replaced by a single NumPy expression. NumPy expressions operate elementwise and hence the above expression works. It basically computes the four point average. If you have gone through the NumPy tutorial and played with `NumPy` a bit you should be able to understand how this works. The beauty of the expression is that its completely done in C. This makes the computation *much* faster. For a quick comparison here are some numbers for a single iteration on a 500x500 grid. On a PIII 450Mhz with 192 MB RAM, the above takes about 0.3 seconds whereas the previous one takes around 15 seconds. This is close to a 50 fold speed increase. You will also note a few things.

1. We cannot compute the error the way we did earlier inside the for loop. We need to make a copy of the data and then use the computeError function to do this. This costs us memory and is not very pretty. This is certainly a limitation but is worth a 50 fold speed increase.
2. The expression will use temporaries. Hence, during one iteration, the computed values at an already computed location will not be used during the iteration. For instance, in the original for loop, once the value of u[1,1] is computed, the next value for u[1,2] will use the newly computed u[1,1] and not the old one. However, since the NumPy expression uses temporaries internally, only the old value of u[1,1] will be used. This is not a serious issue in this case because it is known that even when this happens the algorithm will converge (but in twice as much time, which reduces the benefit by a factor of 2, which still leaves us with a 25 fold increase).

Apart from these two issues its clear that using NumPy boosts speed tremendously. We will now use the amazing weave package to speed this up further.

Using weave.blitz

The NumPy expression can be speeded up quite a bit if we use weave.blitz. Here is the new function.

```
# import necessary modules and functions
from scipy import weave
# ...
    def blitzTimeStep(self, dt=0.0):
        """Takes a time step using a NumPy expression that has beer
        blitzed using weave."""
        g = self.grid
        dx2, dy2 = g.dx**2, g.dy**2
        dnr_inv = 0.5/(dx2 + dy2)
        u = g.u
        g.old_u = u.copy()

        # The actual iteration
        expr = "u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2
               "(u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv"
        weave.blitz(expr, check_size=0)

        return g.computeError()
```

If you notice, the only thing that has changed is that we put quotes around the original numeric expression and call this string ‘expr’ and then invoke weave.blitz. The ‘check_size’ keyword when set to 1 does a few sanity checks and is to be used when you are debugging your code. However, for pure speed it is wise to set it to 0. This time when we time the code for a 500x500 array for a single iteration it takes only about 0.1 seconds which is about a three fold increase! There are again a few things to note.

1. The first time this method is called, it will take a long while to do some magic behind your back. The next time it is called, it will run immediately. More details on this are in the weave documentation. Basically, weave.blitz converts the NumPy expression into C++ code and uses blitz++ for the array expression, builds a Python module, stores it in a special place and invokes that the next time the function call is made.
2. Again we need to use a temporary array to compute the error.
3. blitz does *not* use temporaries for the computation and therefore behaves more like the original (slow) for loop in that the computed values are re-used immediately.

Apart from these points, the results are identical as compared to the original for loop. It’s only about 170 times faster than the original code! We will now look at yet another way to speed up our original for loop. Enter weave.inline!

Using weave.inline

Inline allows one to embed C or C++ code directly into your Python code. Here is a simple version of an inlined version of the code.

```
from scipy.weave import converters
# ...
    def inlineTimeStep(self, dt=0.0):
        """Takes a time step using inlined C code -- this version u
        blitz arrays."""
        g = self.grid
        nx, ny = g.u.shape
        dx2, dy2 = g.dx**2, g.dy**2
        dnr_inv = 0.5/(dx2 + dy2)
        u = g.u

        code = """
               #line 120 "laplace.py" (This is only useful for debu
               double tmp, err, diff;
               err = 0.0;
               for (int i=1; i<nx-1; ++i) {
                   for (int j=1; j<ny-1; ++j) {
                       tmp = u(i,j);
                       u(i,j) = ((u(i-1,j) + u(i+1,j))*dy2 +
                                 (u(i,j-1) + u(i,j+1))*dx2)*dnr_inv
                       diff = u(i,j) - tmp;
                       err += diff*diff;
                   }
              }
               return_val = sqrt(err);
               """
        # compiler keyword only needed on windows with MSVC instal
        err = weave.inline(code,
                           ['u', 'dx2', 'dy2', 'dnr_inv', 'nx', 'ny
                           type_converters=converters.blitz,
                           compiler = 'gcc')
        return err
```

The code itself looks very straightforward (which is what makes inline so cool). The inline function call arguments are all self explanatory. The line with ‘#line 120 ...’ is only used, for debugging and doesn’t affect the speed in anyway. Again the first time you run this function it takes a long while to do something behind the scenes and the next time it blazes away. This time notice that we have far more flexibility inside our loop and can easily compute an error term without a need for temporary arrays. Timing this version results in a time for a 500x500 array of a mere 0.04 seconds per iteration! This corresponds to a whopping 375 fold speed increase over the plain old for loop. And remember we haven’t sacrificed any of Python’s incredible flexibility! This loop contains code that looks very nice but if we want to we can speed things up further by writing a little dirty code. We won’t get into that here but it suffices to say that its possible to get a further factor of two speed up by using a different approach. The code for this basically does pointer arithmetic on the NumPy array data instead of using blitz++ arrays. This code was contributed by Eric Jones. The source code accompanying this article contains this code.

Next, we look at how it is possible to easily implement the loop inside Fortran and call it from Python by using f2py.

Using f2py

f2py is an amazing utility that lets you easily call Fortran functions from Python. First we will write a small Fortran77 subroutine to do our calculation. Here is the code.

```
c File flaplace.f
      subroutine timestep(u,n,m,dx,dy,error)
      double precision u(n,m)
      double precision dx,dy,dx2,dy2,dnr_inv,tmp,diff
      integer n,m,i,j
cf2py intent(in) :: dx,dy
cf2py intent(in,out) :: u
cf2py intent(out) :: error
cf2py intent(hide) :: n,m
      dx2 = dx*dx
      dy2 = dy*dy
      dnr_inv = 0.5d0 / (dx2+dy2)
      error = 0
      do 200,j=2,m-1
         do 100,i=2,n-1
            tmp = u(i,j)
            u(i,j) = ((u(i-1,j) + u(i+1,j))*dy2+
     &           (u(i,j-1) + u(i,j+1))*dx2)*dnr_inv
            diff = u(i,j) - tmp
            error = error + diff*diff
 100     continue
 200  continue
      error = sqrt(error)
      end
```

The lines starting with cf2py are special f2py directives and are documented in f2py. The rest of the code is straightforward for those who know some Fortran. We trivially create a Python module for this using the following command.

```
% f2py -c flaplace.f -m flaplace
```

Here is how the Python side of things looks.

```
import flaplace

    def fortranTimeStep(self, dt=0.0):
        """Takes a time step using a simple fortran module that
        implements the loop in Fortran.  """
        g = self.grid
        g.u, err = flaplace.timestep(g.u, g.dx, g.dy)
        return err
```

Thats it! Hopefully someday scipy.weave will let us do this inline and not require us to write a separate Fortran file. The Fortran code and f2py example were contributed by Pearu Peterson, the author of f2py. Anyway, using this module it takes about 0.029 seconds for a 500x500 grid per iteration! This is about a 500 fold speed increase over the original code.

f2py can also work with more modern Fortran versions. This is useful because Fortran90 has special array features that allow a more compact, nicer and faster code. Here is the same subroutine in Fortran90:

```
! File flaplace90_arrays.f90
subroutine timestep(u,n,m,dx,dy,error)
implicit none
!Pythonic array indices, from 0 to n-1.
real (kind=8), dimension(0:n-1,0:m-1), intent(inout):: u
real (kind=8), intent(in) :: dx,dy
real (kind=8), intent(out) :: error
integer, intent(in) :: n,m
real (kind=8), dimension(0:n-1,0:m-1) :: diff
real (kind=8) :: dx2,dy2,dnr_inv
!f2py intent(in) :: dx,dy
!f2py intent(in,out) :: u
!f2py intent(out) :: error
!f2py intent(hide) :: n,m
dx2 = dx*dx
dy2 = dy*dy
dnr_inv = 0.5d0 / (dx2+dy2)
diff=u
u(1:n-2, 1:m-2) = ((u(0:n-3, 1:m-2) + u(2:n-1, 1:m-2))*dy2 + &
                         (u(1:n-2,0:m-3) + u(1:n-2, 2:m-1))*dx2)*dr
error=sqrt(sum((u-diff)**2))
end subroutine
```

Remark that the operations are performed in a single line, very similar to Numpy. The compilation step is exactly the same. In a 1000x1000 grid, this code is 2.2 times faster than the fortran77 loops.

The source tarball (perfpy_2.tgz) also contains a Fortran95 subroutine using the forall construct, which has a very similar performance.

Using Pyrex/Cython

We also implemented the timeStep function in Pyrex using the code from the fast inline version. The Pyrex sources are a little longer than the weave, blitz or Fortran code since we have to expose the NumPy array structure. The basic function looks like this.

Travis Oliphant wrote a Cython version of the example.

```
def pyrexTimeStep(ndarray u, double dx, double dy):
    if chr(u.descr.type) <> "d":
        raise TypeError("Double array required")
    if u.nd <> 2:
        raise ValueError("2 dimensional array required")
    cdef int nx, ny
    cdef double dx2, dy2, dnr_inv, err
    cdef double *elem

    nx = u.dimensions[0]
    ny = u.dimensions[1]
    dx2, dy2 = dx**2, dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    elem = u.data

    err = 0.0
    cdef int i, j
    cdef double *uc, *uu, *ud, *ul, *ur
    cdef double diff, tmp
    for i from 1 <= i < nx-1:
        uc = elem + i*ny + 1
        ur = elem + i*ny + 2
        ul = elem + i*ny
        uu = elem + (i+1)*ny + 1
        ud = elem + (i-1)*ny + 1

        for j from 1 <= j < ny-1:
            tmp = uc[0]
            uc[0] = ((ul[0] + ur[0])*dy2 +
                     (uu[0] + ud[0])*dx2)*dnr_inv
            diff = uc[0] - tmp
            err = err + diff*diff
            uc = uc + 1; ur = ur + 1;  ul = ul + 1
            uu = uu + 1; ud = ud + 1

    return sqrt(err)
```

The function looks long but is not too hard to write. It is also possible to write without doing the pointer arithmetic by providing convenient functions to access the array. However, the code shown above is fast. The sources provided with this

article contains the complete Pyrex file and also a setup.py script to build it. Timing this version, we find that this version is as fast as the fast inlined version and takes only 0.025 seconds.

Using Matlab and Octave

We have implemented the Numeric version in Matlab and Octave (laplace.m) and run the tests on a different computer (hence the “estimate” values in the table below). We have found that no significant speed-up is obtained in Matlab, while Octave runs twice slower than `NumPy` . Detailed graphs can be found here.

An implementation in C++

Finally, for comparison we implemented this in simple C++ (nothing fancy) without any Python. One would expect that the C++ code would be faster but surprisingly, not by much! Given the fact that it’s so easy to develop with Python, this speed reduction is not very significant.

A final comparison

Here are some timing results for a 500x500 grid for 100 iterations. Note that we also have a comparison of results of using the slow Python version along with Psyco.

Type of solution |Time taken (sec)—|—Python (estimate) |1500.0Python + Psyco (estimate) |1138.0Python + `NumPy` Expression | 29.3Blitz | 9.5Inline | 4.3Fast Inline | 2.3Python/Fortran | 2.9Pyrex | 2.5Matlab (estimate) | 29.0Octave (estimate) | 60.0Pure C++ | 2.16

This is pretty amazing considering the flexibility and power of Python.

Download the source code for this guide here: `perfpy_2.tgz`

View the complete Python code for the example: `laplace.py`

View the complete Matlab/Octave code for the example: `laplace.m`

Attachments

- `laplace.m`
- `laplace.py`
- `perfpy_2.tgz`

Parallel Programming with numpy and scipy

Multiprocessor and multicore machines are becoming more common, and it would be nice to take advantage of them to make your code run faster. numpy/scipy are not perfect in this area, but there are some things you can do. The best way to make use of a parallel processing system depend on the task you're doing and on the parallel system you're using. If you have a big complicated job or a cluster of machines, taking full advantage will require much thought. But many tasks can be parallelized in a fairly simple way.

As the saying goes, "premature optimization is the root of all evil". Using a multicore machine will provide at best a speedup by a factor of the number of cores available. Get your code working first, before even thinking about parallelization. Then ask yourself whether your code actually needs to be any faster. Don't embark on the bug-strewn path of parallelization unless you have to.

Simple parallelization

Break your job into smaller jobs and run them simultaneously

For example, if you are analyzing data from a pulsar survey, and you have thousands of beams to analyze, each taking a day, the simplest (and probably most efficient) way to parallelize the task is to simply run each beam as a job. Machines with two processors can just run two jobs. No need to worry about locking or communication; no need to write code that knows it's running in parallel. You can have issues if the processes each need as much memory as your machine has, or if they are both IO heavy, but in general this is a simple and efficient way to parallelize your code - if it works. Not all tasks divide up this nicely. If your goal is to process a single image, it's not clear how to do this without a lot of work.

Use parallel primitives

One of the great strengths of numpy is that you can express array operations very cleanly. For example to compute the product of the matrix A and the matrix B, you just do:

```
>>> C = numpy.dot(A,B)
```

Not only is this simple and clear to read and write, since numpy knows you want to do a matrix dot product it can use an optimized implementation obtained as part of “BLAS” (the Basic Linear Algebra Subroutines). This will normally be a library carefully tuned to run as fast as possible on your hardware by taking advantage of cache memory and assembler implementation. But many architectures now have a BLAS that also takes advantage of a multicore machine. If your numpy/scipy is compiled using one of these, then dot() will be computed in parallel (if this is faster) without you doing anything. Similarly for other matrix operations, like inversion, singular value decomposition, determinant, and so on. For example, the open source library ATLAS allows compile time selection of the level of parallelism (number of threads). The proprietary MKL library from Intel offers the possibility to chose the level of parallelism at runtime. There is also the GOTO library that allow run-time selection of the level of parallelism. This is a commercial product but the source code is distributed free for academic use.

Finally, scipy/numpy does not parallelize operations like

```
>>> A = B + C
>>> A = numpy.sin(B)
>>> A = scipy.stats.norm.isf(B)
```

These operations run sequentially, taking no advantage of multicore machines (but see below). In principle, this could be changed without too much work. OpenMP is an extension to the C language which allows compilers to produce parallelizing code for appropriately-annotated loops (and other things). If someone sat down and annotated a few core loops in numpy (and possibly in scipy), and if one then compiled numpy/scipy with OpenMP turned on, all three of the above would automatically be run in parallel. Of course, in reality one would want to have some runtime control - for example, one might want to turn off automatic parallelization if one were planning to run several jobs on the same multiprocessor machine.

Write multithreaded or multiprocess code

Sometimes you can see how to break your problem into several parallel tasks, but those tasks need some kind of synchronization or communication. For example, you might have a list of jobs that can be run in parallel, but you need to gather all their results, do some summary calculation, then launch another batch of parallel jobs. There are two approaches to doing this in Python, using either multiple threads) or processes).

Threads

Threads are generally ‘lighter’ than processes, and can be created, destroyed and switched between faster than processes. They are normally preferred for taking advantage of multicore systems. However, multithreading with python has a key

limitation; the Global Interpreter Lock (GIL). For various reasons (a quick web search will turn up copious discussion, not to say argument, over why this exists and whether it's a good idea), python is implemented in such a way that only one thread can be accessing the interpreter at a time. This basically means only one thread can be running python code at a time. This almost means that you don't take any advantage of parallel processing at all. The exceptions are few but important: while a thread is waiting for IO (for you to type something, say, or for something to come in the network) python releases the GIL so other threads can run. And, more importantly for us, while numpy is doing an array operation, python also releases the GIL. Thus if you tell one thread to do:

```
>>> print "%s  %s  %s  %s and %s" %( ("spam",) *3 + ("eggs",) + ("s
>>> A = B + C
>>> print A
```

During the print operations and the % formatting operation, no other thread can execute. But during the A = B + C, another thread can run - and if you've written your code in a numpy style, much of the calculation will be done in a few array operations like A = B + C. Thus you can actually get a speedup from using multiple threads.

The python threading module is part of the standard library and provides tools for multithreading. Given the limitations discussed above, it may not be worth carefully rewriting your code in a multithreaded architecture. But sometimes you can do multithreading with little effort, and in these cases it can be worth it. See `Cookbook/Multithreading` for one example. This recipe provides a thread Pool() interface with the same API as that found for processes (below) which might also be of interest. There is also the ThreadPo ol module which is quite similar.

Processes

One way to overcome the limitations of the GIL discussed above is to use multiple full processes instead of threads. Each process has it's own GIL, so they do not block each other in the same way that threads do. From python 2.6, the standard library includes a multiprocessing module, with the same interface as the threading module. For earlier versions of Python, this is available as the processing module (a backport of the multiprocessing module of python 2.6 for python 2.4 and 2.5 is in the works here: multiprocessing). It is possible to share memory between processes, including numpy arrays. This allows most of the benefits of threading without the problems of the GIL. It also provides a simple Pool() interface, which features map and apply commands similar to those found in the `Cookbook/Multithreading` example.

Comparison

Here is a very basic comparison which illustrates the effect of the GIL (on a dual core machine).

```
import numpy as np
import math
def f(x):
    print x
    y = [1]*10000000
    [math.exp(i) for i in y]
def g(x):
    print x
    y = np.ones(10000000)
    np.exp(y)

from handythread import foreach
from processing import Pool
from timings import f,g
def fornorm(f,l):
    for i in l:
        f(i)
time fornorm(g,range(100))
time fornorm(f,range(10))
time foreach(g,range(100),threads=2)
time foreach(f,range(10),threads=2)
p = Pool(2)
time p.map(g,range(100))
time p.map(f,range(100))
```

| 100 *g()* | *10* f()—|—|—normal | `43.5s` | `48s` 2 threads | `31s` | `71.5s` 2 processes | `27s` | `31.23`

For function `f()`, which does not release the GIL, threading actually performs worse than serial code, presumably due to the overhead of context switching. However, using 2 processes does provide a significant speedup. For function `g()` which uses numpy and releases the GIL, both threads and processes provide a significant speed up, although multiprocesses is slightly faster.

Sophisticated parallelization

If you need sophisticated parallelism - you have a computing cluster, say, and your jobs need to communicate with each other frequently - you will need to start thinking about real parallel programming. This is a subject for graduate courses in computer science, and I'm not going to address it here. But there are some python tools you can use to implement the things you learn in that graduate course. (I am perhaps exaggerating - some parallelization is not that difficult, and some of these tools make it fairly easy. But do realize that parallel code is much more difficult to write and debug than serial code.)

- IPython1
- `mpi4py`
- parallel python
- POSH

Scientific GUIs

- wxPython dialogs

wxPython dialogs

Something I enjoy in matlab is the ease in which simple file selector dialogs, and status bars can be made. Now that I use nothing but scipy, I have wanted to have simliar functions for my scipy code. Thanks to the great reference “wxPython in Action” I have learned some of the basics again, with the promise of making very fancy GUIs if I ever find the urge! (Check out the sample chapters, they have given the entire section on making dialogs, which is how I initially got started with wxPython).

File Selector Dialog

I often write simple translation scripts that convert some data, into another form. I like to use these for a series of data, and share them with some coworkers who do not program. the wxPython FileSelector function comes to the rescue.

```
import wx

# setup the GUI main loop
app = wx.App()

filename = wx.FileSelector()
```

With this basic code, filename contains a string pathname (may be unicode depending on your installation of wxPython, more on this latter) of the selected file.

Some of the spiffing up will include using the current directory the script was started in, we do this easily

```
import wx
import os

# setup the GUI main loop
app = wx.App()

filename = wx.FileSelector(default_path=os.getcwd())
```

If one runs such script repeatedly, it might be a good idea to do some basic clean-up after each run.

```
# ...
app.Destroy()
```

Scientific Scripts

FDTD Algorithm Applied to the Schrödinger Equation

The code below illustrates the use of the The One-Dimensional Finite-Difference Time-Domain (FDTD) algorithm to solve the one-dimensional Schrödinger equation for simple potentials. It only requires Numpy and Matplotlib.

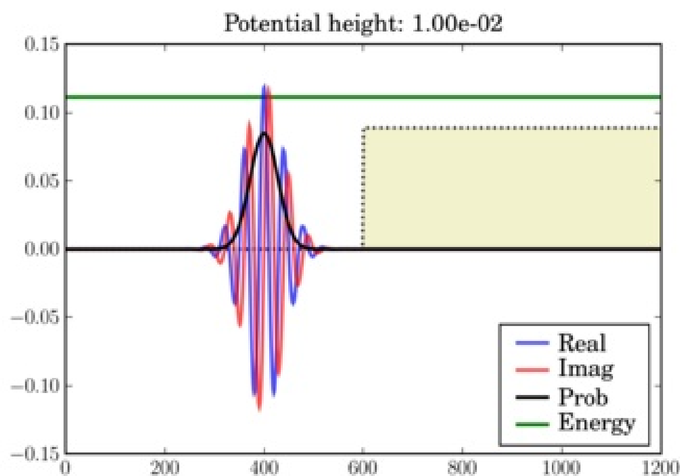
All the mathematical details are described in this PDF: `Schrodinger_FDTD.pdf`

Examples

In these figures the potential is shaded in arbitrary units in yellow, while the total energy of the wavepacket is plotted as a green line, in the same units as the potential. So while the energy units are *not* those on the left axis, both energy plots use the same units and can thus be validly compared relative to one another.

Depending on the particle energy, the yellow region may be classically forbidden (when the green line is inside the yellow region).

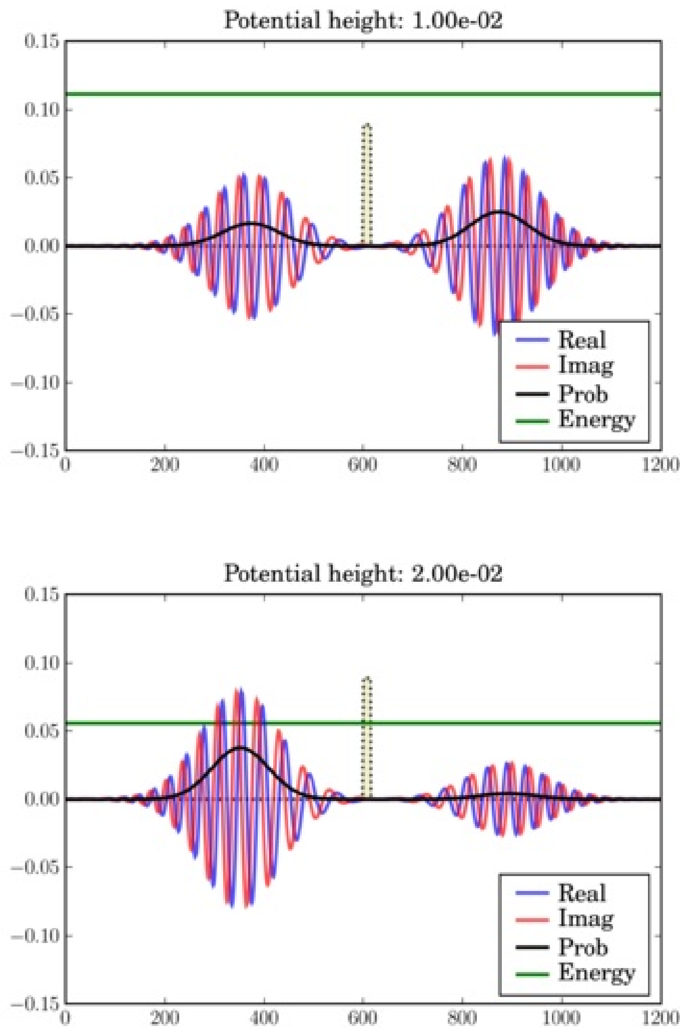
The wavepacket starts at t=0 as (step potential shown):



And at the end of the simulation it can look like this, depending on the actual potential height:



This illustrates the tunneling through a thin barrier, depending on the barrier height. In the second case, a classical particle would completely bounce off since its energy is lower than the potential barrier:



Code

```
#=======================================================================
#
#                Quantum Mechanical Simulation using Finite-Differen
#                Time-Domain (FDTD) Method
#
#        This script simulates a probability wave in the presence of
#        potentials.  The simulation is c arried out by using the FD
#        applied to the Schrodinger equation.  The program is intend
#        a demonstration of the FDTD algorithm and can be used as an
#        aid for quantum mechanics and numerical methods.  The simul
#        parameters are defined in the code constants and can be fre
#        manipulated to see different behaviors.
#
#        NOTES
#
#        The probability density plots are amplified by a factor for
#        purposes.  The psi_p quanity contains the actual probabilit
#        without any rescaling.
```

```
#
#       BEWARE: The time step, dt, has strict requirements or else
#       simulation becomes unstable.
#
#       The code has three built-in potential functions for demonst
#
#       1) Constant potential: Demonstrates a free particle with di
#
#       2) Step potential: Demonstrates transmission and reflection
#
#       3) Potential barrier: Demonstrates tunneling.
#
#       By tweaking the height of the potential (V0 below) as well
#       barrier thickness (THCK below), you can see different behav
#       reflection with no noticeable transmission, transmission an
#       reflection, or mostly transmission with tunneling.
#
#               This script requires pylab and numpy to be installe
#               Python or else it will not run.
#
#==================================================================
# Author:  James Nagel <nagel@mers.byu.edu>
#          5/25/07
#
# Updates by Fernando Perez <Fernando.Perez@colorado.edu>, 7/28/07
#==================================================================
#  Numerical and plotting libraries
import numpy as np
import pylab
# Set pylab to interactive mode so plots update when run outside ip
pylab.ion()
#==================================================================
# Utility functions
#  Defines a quick Gaussian pulse function to act as an envelope to
#  function.
def Gaussian(x,t,sigma):
    """  A Gaussian curve.
 x = Variable
 t = time shift
 sigma = standard deviation      """
    return np.exp(-(x-t)**2/(2*sigma**2))
def free(npts):
    "Free particle."
    return np.zeros(npts)
def step(npts,v0):
    "Potential step"
    v = free(npts)
    v[npts/2:] = v0
    return v
def barrier(npts,v0,thickness):
    "Barrier potential"
    v = free(npts)
    v[npts/2:npts/2+thickness] = v0
```

```
    return v
def fillax(x,y,*args,**kw):
    """Fill the space between an array of y values and the x axis.
 All args/kwargs are passed to the pylab.fill function.
 Returns the value of the pylab.fill() call.
 """
    xx = np.concatenate((x,np.array([x[-1],x[0]],x.dtype)))
    yy = np.concatenate((y,np.zeros(2,y.dtype)))
    return pylab.fill(xx, yy, *args,**kw)

#=================================================================
#
#  Simulation Constants.  Be sure to include decimal points on appr
#  variables so they become floats instead of integers.
#
N    = 1200     #  Number of spatial points.
T    = 5*N      #  Number of time steps.  5*N is a nice value for t
                #  before anything reaches the boundaries.
Tp   = 50       #  Number of time steps to increment before updatin
dx   = 1.0e0    #  Spatial resolution
m    = 1.0e0    #  Particle mass
hbar = 1.0e0    #  Plank's constant
X    = dx*np.linspace(0,N,N)        #  Spatial axis.
# Potential parameters.  By playing with the type of potential and
# and thickness (for barriers), you'll see the various transmission
# regimes of quantum mechanical tunneling.
V0   = 1.0e-2   #  Potential amplitude (used for steps and barriers
THCK = 15       # "Thickness" of the potential barrier (if appropri
                # V-function is chosen)
# Uncomment the potential type you want to use here:
# Zero potential, packet propagates freely.
#POTENTIAL = 'free'
# Potential step.  The height (V0) of the potential chosen above wi
# the amount of reflection/transmission you'll observe
POTENTIAL = 'step'
# Potential barrier.  Note that BOTH the potential height (V0) and
# of the barrier (THCK) affect the amount of tunneling vs reflectio
# observe.
#POTENTIAL = 'barrier'
#  Initial wave function constants
sigma = 40.0 # Standard deviation on the Gaussian envelope (remembe
             #  uncertainty).
x0 = round(N/2) - 5*sigma # Time shift
k0 = np.pi/20 # Wavenumber (note that energy is a function of k)
# Energy for a localized gaussian wavepacket interacting with a loc
# potential (so the interaction term can be neglected by computing
# integral over a region where V=0)
E = (hbar**2/2.0/m)*(k0**2+0.5/sigma**2)
#=================================================================
# Code begins
#
# You shouldn't need to change anything below unless you want to ac
# with the numerical algorithm or modify the plotting.
```

```
#
#  Fill in the appropriate potential function (is there a Python ec
#  the SWITCH statement?).
if POTENTIAL=='free':
    V = free(N)
elif POTENTIAL=='step':
    V = step(N,V0)
elif POTENTIAL=='barrier':
    V = barrier(N,V0,THCK)
else:
    raise ValueError("Unrecognized potential type: %s" % POTENTIAL)
#  More simulation parameters.  The maximum stable time step is a f
#  the potential, V.
Vmax = V.max()            #  Maximum potential of the domain.
dt   = hbar/(2*hbar**2/(m*dx**2)+Vmax)         #  Critical time ste
c1   = hbar*dt/(m*dx**2)                       #  Constant coeffici
c2   = 2*dt/hbar                               #  Constant coeffici
c2V  = c2*V  # pre-compute outside of update loop
# Print summary info
print 'One-dimensional Schrodinger equation - time evolution'
print 'Wavepacket energy:   ',E
print 'Potential type:      ',POTENTIAL
print 'Potential height V0: ',V0
print 'Barrier thickness:   ',THCK
#  Wave functions.  Three states represent past, present, and futur
psi_r = np.zeros((3,N)) #  Real
psi_i = np.zeros((3,N)) #  Imaginary
psi_p = np.zeros(N,)   # Observable probability (magnitude-squared
                          #  of the complex wave function).
#  Temporal indexing constants, used for accessing rows of the wave
PA = 0                 #  Past
PR = 1                 #  Present
FU = 2                 #  Future
#  Initialize wave function.  A present-only state will "split" wit
#  wave function propagating to the left and the other half to the
#  Including a "past" state will cause it to propagate one way.
xn = range(1,N/2)
x = X[xn]/dx    #  Normalized position coordinate
gg = Gaussian(x,x0,sigma)
cx = np.cos(k0*x)
sx = np.sin(k0*x)
psi_r[PR,xn] = cx*gg
psi_i[PR,xn] = sx*gg
psi_r[PA,xn] = cx*gg
psi_i[PA,xn] = sx*gg
# Initial normalization of wavefunctions
#   Compute the observable probability.
psi_p = psi_r[PR]**2 + psi_i[PR]**2
#  Normalize the wave functions so that the total probability in th
#  is equal to 1.
P   = dx * psi_p.sum()                      #  Total probability.
nrm = np.sqrt(P)
psi_r /= nrm
```

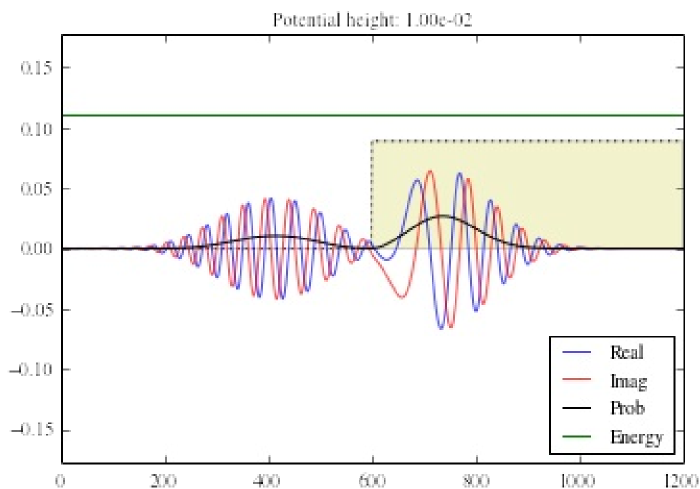
```
psi_i /= nrm
psi_p /= P
#  Initialize the figure and axes.
pylab.figure()
xmin = X.min()
xmax = X.max()
ymax = 1.5*(psi_r[PR]).max()
pylab.axis([xmin,xmax,-ymax,ymax])
#  Initialize the plots with their own line objects.  The figures p
#  faster if you simply update the lines as opposed to redrawing th
#  figure.  For reference, include the potential function as well.
lineR, = pylab.plot(X,psi_r[PR],'b',alpha=0.7,label='Real')
lineI, = pylab.plot(X,psi_i[PR],'r',alpha=0.7,label='Imag')
lineP, = pylab.plot(X,6*psi_p,'k',label='Prob')
pylab.title('Potential height: %.2e' % V0)
# For non-zero potentials, plot them and shade the classically forb
# in light red, as well as drawing a green line at the wavepacket's
# energy, in the same units the potential is being plotted.
if Vmax !=0 :
    # Scaling factor for energies, so they fit in the same plot as
    # wavefunctions
    Efac = ymax/2.0/Vmax
    V_plot = V*Efac
    pylab.plot(X,V_plot,':k',zorder=0)   #  Potential line.
    fillax(X,V_plot, facecolor='y', alpha=0.2,zorder=0)
    # Plot the wavefunction energy, in the same scale as the potent
    pylab.axhline(E*Efac,color='g',label='Energy',zorder=1)
pylab.legend(loc='lower right')
pylab.draw()
# I think there's a problem with pylab, because it resets the xlim
# plotting the E line.  Fix it back manually.
pylab.xlim(xmin,xmax)
#  Direct index assignment is MUCH faster than using a spatial FOR
#  these constants are used in the update equations.  Remember that
#  zero-based indexing.
IDX1 = range(1,N-1)                             #  psi [ k ]
IDX2 = range(2,N)                               #  psi [ k + 1 ]
IDX3 = range(0,N-2)                             #  psi [ k - 1 ]
for t in range(T+1):
    # Precompute a couple of indexing constants, this speeds up the
    psi_rPR = psi_r[PR]
    psi_iPR = psi_i[PR]
    #  Apply the update equations.
    psi_i[FU,IDX1] = psi_i[PA,IDX1] + \
                      c1*(psi_rPR[IDX2] - 2*psi_rPR[IDX1] +
                          psi_rPR[IDX3])
    psi_i[FU] -= c2V*psi_r[PR]

    psi_r[FU,IDX1] = psi_r[PA,IDX1] - \
                      c1*(psi_iPR[IDX2] - 2*psi_iPR[IDX1] +
                          psi_iPR[IDX3])
    psi_r[FU] += c2V*psi_i[PR]
    #  Increment the time steps.  PR -> PA and FU -> PR
```

```
    psi_r[PA] = psi_rPR
    psi_r[PR] = psi_r[FU]
    psi_i[PA] = psi_iPR
    psi_i[PR] = psi_i[FU]
    #  Only plot after a few iterations to make the simulation run
    if t % Tp == 0:
        #  Compute observable probability for the plot.
        psi_p = psi_r[PR]**2 + psi_i[PR]**2
        #  Update the plots.
        lineR.set_ydata(psi_r[PR])
        lineI.set_ydata(psi_i[PR])
        # Note: we plot the probability density amplified by a fact
        # bit easier to see.
        lineP.set_ydata(6*psi_p)

        pylab.draw()
# So the windows don't auto-close at the end if run outside ipythor
pylab.ioff()
pylab.show()
```

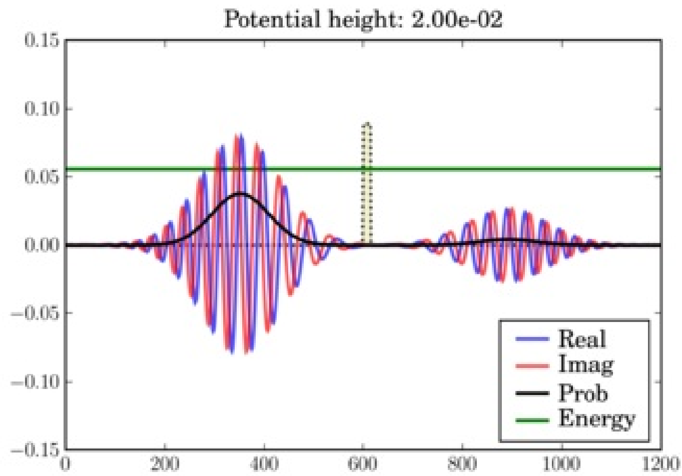
```
One-dimensional Schrodinger equation - time evolution
Wavepacket energy:     0.0124932555014
Potential type:        step
Potential height V0:   0.01
Barrier thickness:     15
```

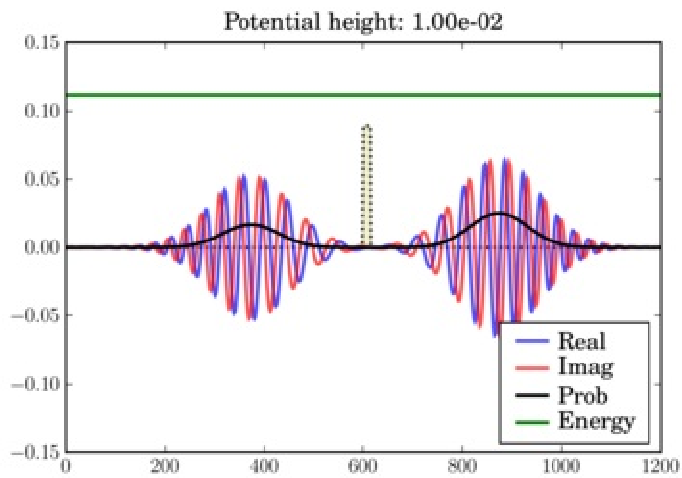


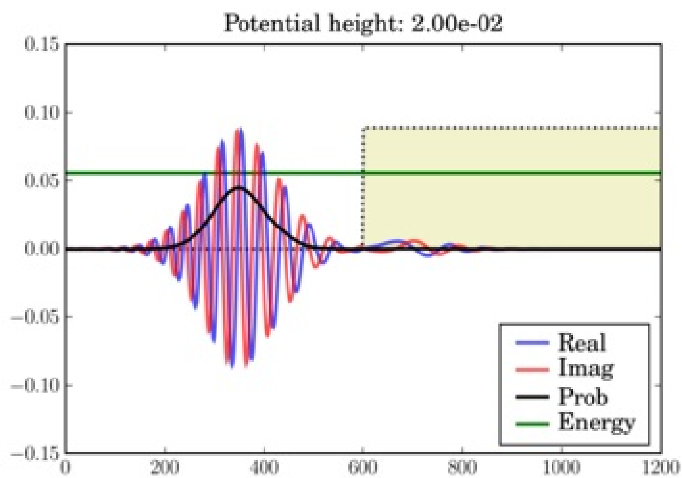
Attachments

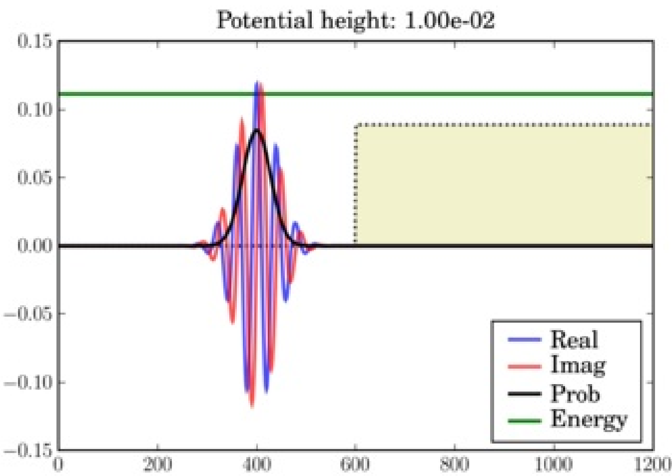
- Schrodinger_FDTD.pdf
- schrod_barrier_hi_sm2.png
- schrod_barrier_lo_sm2.png
- schrod_step_hi_sm2.png

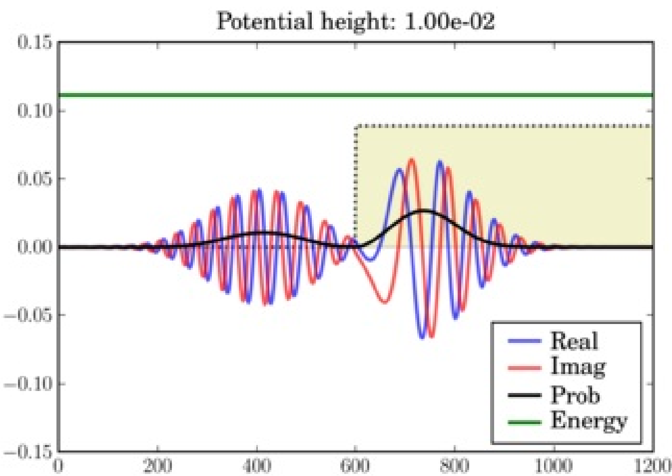
- `schrod_step_init_sm2.png`
- `schrod_step_lo_sm2.png`








Potential height: 1.00e-02
Real
Imag
Prob
Energy


Potential height: 1.00e-02
Real
Imag
Prob
Energy

Using NumPy With Other Languages (Advanced)

- C extensions
- Ctypes
- F2py
- Inline Weave With Basic Array Conversion (no Blitz)
- Pyrex And NumPy
- SWIG Numpy examples
- SWIG and Numpy
- SWIG memory deallocation
- f2py and numpy

C extensions

Skeleton

`extmodule.h` :

```
#ifndef EXTMODULE_H
#define EXTMODULE_H
#ifdef __cplusplus
extern “C” {
#endif
/* Python.h must be included before everything else */
#include “Python.h”
/* include system headers here */
#if !defined(EXTMODULE_IMPORT_ARRAY)
#define NO_IMPORT_ARRAY
#endif
#include “numpy/arrayobject.h”
#ifdef __cplusplus
}
#endif
#endif
```

Note that `PY_ARRAY_UNIQUE_SYMBOL` must be set to a unique value for each extension module. But, you actually don’t need to set it at all unless you are going to compile an extension module that uses two different source files

`extmodule.c` :

```
#define EXTMODULE_IMPORT_ARRAY
#include "extmodule.h"
#undef EXTMODULE_IMPORT_ARRAY
static PyObject* FooError;
static PyObject* Ext_Foo(PyObject* obj, PyObject* args) {
    Py_INCREF(Py_None); return Py_None;
}
static PyMethodDef methods[] = {
    {"foo", (PyCFunction) Ext_Foo, METH_VARARGS, ""}, {NULL, NULL,
};
PyMODINIT_FUNC init_extmodule() {
    PyObject* m;
    m = Py_InitModule("_extmodule", methods);
    import_array();
    SVMError = PyErr_NewException("_extmodule.FooError", NULL, NULL
    Py_INCREF(FooError);
    PyModule_AddObject(m, "FooError", FooError);
}
```

If your extension module is contained in a single source file then you can get rid of extmodule.h entirely and replace the first part of extmodule.c with

```
#inlude "Python.h"
#include "numpy/arrayobject.h"
/* remainder of extmodule.c after here */
```

Debugging C Extensions on Windows

Debugging C extensions on Windows can be tricky. If you compile your extension code in debug mode, you have to link against the debug version of the Python library, e.g. `Python24_d.lib`. When building with Visual Studio, this is taken care of by a pragma in `Python24.h`. If you force the compiler to link debug code against the release library, you will probably get the following errors (especially when compiling SWIG wrapped codes):

```
extmodule.obj : error LNK2019: unresolved external symbol __imp___F
extmodule.obj : error LNK2019: unresolved external symbol __imp___F
extmodule.obj : error LNK2001: unresolved external symbol __imp___F
extmodule.obj : error LNK2019: unresolved external symbol __imp___F
extmodule.obj : error LNK2019: unresolved external symbol __imp___F
extmodule.obj : error LNK2019: unresolved external symbol __imp__Py
```

However, now you also need a debug build of the Python interpreter if you want to import this debuggable extension module. Now you also need debug builds of every other extension module you use. Clearly, this can take some time to get sorted out.

An alternative is to build your library code as a debug DLL. This way, you can at least that your extension module is passing the right data to the library code you are wrapping.

As an aside, it seems that the MinGW GCC compiler doesn't produce debug symbols that are understood by the Visual Studio debugger.

Valgrind

To develop a stable extension module, it is essential that you check the memory allocation and memory accesses done by your C code. On Linux, you can use Valgrind. On Windows, you could try a commercial tool such as Rational PurifyPlus.

Before using Valgrind, make sure your extension module is compiled with the `-g` switch to GCC so that you can get useful stack traces when errors are detected.

Then put the following in a shell script, say `valgrind_py.sh` :

```
#!/bin/sh valgrind

-tool=memcheck -leak-check=yes -error-limit=no -suppressions=valgr
```

`valgrind-python.supp` suppresses some warnings caused by the Python code. You can find the suppression file for Python 2.4 in the Python SVN repository. See also README.valgrind in the same location. Some of the suppressions are commented out by default. Enable them by removing the # comment markers.

Execute `chmod +x valgrind_py.sh` and run it as `./valgrind_py.sh test_extmodule.py` .

Documentation

- Extending and Embedding the Python Interpreter (read this first)
- Python/C API Reference Manual (then browse through this)
- Chapter 13 of Guide to NumPy describes the complete API
- Chapter 14 deals with extending !NumPy (make sure you have the edition dated March 15, 2006 or later)

Examples

- ndimage in the SciPy SVN repository
- `NumPy arrays` (collection of small examples)

Mailing List Threads

- Freeing memory allocated in C
- port C/C++ matlab mexing code to numpy
- newbie for writing numpy/scipy extensions
- Array data and struct alignment

Ctypes

Introduction

ctypes is an advanced Foreign Function Interface package for Python 2.3 and higher. It is included in the standard library for Python 2.5.

ctypes allows to call functions exposed from DLLs/shared libraries and has extensive facilities to create, access and manipulate simple and complicated C data types in Python - in other words: wrap libraries in pure Python. It is even possible to implement C callback functions in pure Python.

ctypes also includes a code generator tool chain which allows automatic creation of library wrappers from C header files. ctypes works on Windows, Mac OS X, Linux, Solaris, FreeBSD, OpenBSD and other systems.

Ensure that you have at least ctypes version 1.0.1 or later.

Other possibilities to call or run C code in python include: SWIG, Cython, Weave, cffi, etc.

Getting Started with ctypes

The ctypes tutorial and the ctypes documentation for Python provide extensive information on getting started with ctypes.

Assuming you've built a library called `foo.dll` or `libfoo.so` containing a function called `bar` that takes a pointer to a buffer of doubles and an int as arguments and returns an int, the following code should get you up and running. The following sections cover some possible build scripts, C code and Python code.

If you would like to build your DLL/shared library with distutils, take a look at the !SharedLibrary distutils extension included with OOF2. This should probably be included in numpy.distutils at some point.

Nmake Makefile (Windows)

Run nmake inside the Visual Studio Command Prompt to build with the following file.

You should be able to build the DLL with any version of the Visual Studio compiler regardless of the compiler used to compile Python. Keep in mind that you shouldn't allocate/deallocate memory across different debug/release and single-threaded/multi-threaded runtimes or operate on FILE*s from different runtimes.

```
CXX = cl.exe
LINK = link.exe

CPPFLAGS = -D_WIN32 -D_USRDLL -DFOO_DLL -DFOO_EXPORTS
CXXFLAGSALL = -nologo -EHsc -GS -W3 -Wp64 $(CPPFLAGS)
CXXFLAGSDBG = -MDd -Od -Z7 -RTCcsu
CXXFLAGSOPT = -MD -O2
#CXXFLAGS = $(CXXFLAGSALL) $(CXXFLAGSDBG)
CXXFLAGS = $(CXXFLAGSALL) $(CXXFLAGSOPT)

LINKFLAGSALL = /nologo /DLL
LINKFLAGSDBG = /DEBUG
LINKFLAGSOPT =
#LINKFLAGS = $(LINKFLAGSALL) $(LINKFLAGSDBG)
LINKFLAGS = $(LINKFLAGSALL) $(LINKFLAGSOPT)

all: foo.dll

foo.dll: foo.obj
    $(LINK) $(LINKFLAGS) foo.obj /OUT:foo.dll

svm.obj: svm.cpp svm.h
    $(CXX) $(CXXFLAGS) -c foo.cpp

clean:
    -erase /Q *.obj *.dll *.exp *.lib
```

SConstruct (GCC)

You can use the following file with SCons to build a shared library.

```
env = Environment()
env.Replace(CFLAGS=['-O2','-Wall','-ansi','-pedantic'])
env.SharedLibrary('foo', ['foo.cpp'])
```

foo.cpp

```
#include <stdio.h>

#ifdef FOO_DLL
#ifdef FOO_EXPORTS
#define FOO_API __declspec(dllexport)
#else
#define FOO_API __declspec(dllimport)
#endif /* FOO_EXPORTS */
#else
#define FOO_API extern /* XXX confirm this */
#endif /* FOO_DLL */

#ifdef __cplusplus
extern "C" {
#endif

extern FOO_API int bar(double* data, int len) {
   int i;
   printf("data = %p\n", (void*) data);
   for (i = 0; i < len; i++) {
      printf("data[%d] = %f\n", i, data[i]);
   }
   printf("len = %d\n", len);
   return len + 1;
}

#ifdef __cplusplus
}
#endif
```

When building the DLL for foo on Windows, define `FOO_DLL` and `FOO_EXPORTS` (this is what you want to do when building a DLL for use with ctypes). When linking against the DLL, define `FOO_DLL`. When linking against a static library that contains foo, or when including foo in an executable, don't define anything.

If you're unclear about what is for, read section 3 of the C++ dlopen mini HOWTO. This allows you to write function wrappers with C linkage on top of a bunch of C++ classes so that you can use them with ctypes. Alternatively, you might prefer to write C code.

foo.py

```
import numpy as N
import ctypes as C
_foo = N.ctypeslib.load_library('libfoo', '.')
_foo.bar.restype = C.c_int
_foo.bar.argtypes = [C.POINTER(C.c_double), C.c_int]
def bar(x):
    return _foo.bar(x.ctypes.data_as(C.POINTER(C.c_double)), len(x)
x = N.random.randn(10)
n = bar(x)
```

NumPy arrays' ctypes property

A ctypes property was recently added to NumPy arrays:

```
In [18]: x = N.random.randn(2,3,4)

In [19]: x.ctypes.data
Out[19]: c_void_p(14394256)

In [21]: x.ctypes.data_as(ctypes.POINTER(c_double))

In [24]: x.ctypes.shape
Out[24]: <ctypes._endian.c_long_Array_3 object at 0x00DEF2B0>

In [25]: x.ctypes.shape[:3]
Out[25]: [2, 3, 4]

In [26]: x.ctypes.strides
Out[26]: <ctypes._endian.c_long_Array_3 object at 0x00DEF300>

In [27]: x.ctypes.strides[:3]
Out[27]: [96, 32, 8]
```

In general, a C function might take a pointer to the array's data, an integer indicating the number of array dimensions, (pass the value of the ndim property here) and two int pointers to the shapes and stride information.

If your C function assumes contiguous storage, you might want to wrap it with a Python function that calls !NumPy's `ascontiguousarray` function on all the input arrays.

NumPy's ndpointer with ctypes argtypes

Starting with ctypes 0.9.9.9, any class implementing the from_param method can be used in the argtypes list of a function. Before ctypes calls a C function, it uses the argtypes list to check each parameter.

Using !NumPy's ndpointer function, some very useful argtypes classes can be constructed, for example:

```
from numpy.ctypeslib import ndpointer
arg1 = ndpointer(dtype='<f4')
arg2 = ndpointer(ndim=2)
arg3 = ndpointer(shape=(10,10))
arg4 = ndpointer(flags='CONTIGUOUS,ALIGNED')
# or any combination of the above
arg5 = ndpointer(dtype='>i4', flags='CONTIGUOUS')
func.argtypes = [arg1,arg2,arg3,arg4,arg5]
```

Now, if an argument doesn't meet the requirements, a !TypeError is raised. This allows one to make sure that arrays passed to the C function is in a form that the function can handle.

See also the mailing list thread on ctypes and ndpointer.

Dynamic allocation through callbacks

ctypes supports the idea of callbacks, allowing C code to call back into Python through a function pointer. This is possible because ctypes releases the Python Global Interpreter Lock (GIL) before calling the C function.

We can use this feature to allocate !NumPy arrays if and when we need a buffer for C code to operate on. This could avoid having to copy data in certain cases. You also don't have to worry about freeing the C data after you're done with it. By allocating your buffers as !NumPy arrays, the Python garbage collector can take care of this.

Python code:

```
from ctypes import *
ALLOCATOR = CFUNCTYPE(c_long, c_int, POINTER(c_int))
# load your library as lib
lib.baz.restype = None
lib.baz.argtypes = [c_float, c_int, ALLOCATOR]
```

This isn't the prettiest way to define the allocator (I'm also not sure if c_long is the right return type), but there are a few bugs in ctypes that seem to make this the only way at present. Eventually, we'd like to write the allocator like this (but it doesn't work yet):

```
from numpy.ctypeslib import ndpointer
ALLOCATOR = CFUNCTYPE(ndpointer('f4'), c_int, POINTER(c_int))
```

The following also seems to cause problems:

```
ALLOCATOR = CFUNCTYPE(POINTER(c_float), c_int, POINTER(c_int))
ALLOCATOR = CFUNCTYPE(c_void_p, c_int, POINTER(c_int))
ALLOCATOR = CFUNCTYPE(None, c_int, POINTER(c_int), POINTER(c_void_p
```

Possible failures include a !SystemError exception being raised, the interpreter crashing or the interpreter hanging. Check these mailing list threads for more details: *Pointer-to-pointer unchanged when assigning in callback* Hang with callback returning POINTER(c_float) * Error with callback function and as_parameter with NumPy ndpointer

Time for an example. The C code for the example:

```
#ifndef CSPKREC_H
#define CSPKREC_H
#ifdef FOO_DLL
#ifdef FOO_EXPORTS
#define FOO_API __declspec(dllexport)
#else
#define FOO_API __declspec(dllimport)
#endif
#else
#define FOO_API
#endif
#endif
#include <stdio.h>
#ifdef __cplusplus
extern "C" {
#endif

typedef void*(*allocator_t)(int, int*);

extern FOO_API void foo(allocator_t allocator) {
   int dim = 2;
   int shape[] = {2, 3};
   float* data = NULL;
   int i, j;
   printf("foo calling allocator\n");
   data = (float*) allocator(dim, shape);
   printf("allocator returned in foo\n");
   printf("data = 0x%p\n", data);
   for (i = 0; i < shape[0]; i++) {
      for (j = 0; j < shape[1]; j++) {
         *data++ = (i + 1) * (j + 1);
      }
   }
}

#ifdef __cplusplus
}
#endif
```

Check the The Function Pointer Tutorials if you're new to function pointers in C or C++. And the Python code:

```
from ctypes import *
import numpy as N

allocated_arrays = []
def allocate(dim, shape):
    print 'allocate called'
    x = N.zeros(shape[:dim], 'f4')
    allocated_arrays.append(x)
    ptr = x.ctypes.data_as(c_void_p).value
    print hex(ptr)
    print 'allocate returning'
    return ptr

lib = cdll['callback.dll']
lib.foo.restype = None
ALLOCATOR = CFUNCTYPE(c_long, c_int, POINTER(c_int))
lib.foo.argtypes = [ALLOCATOR]

print 'calling foo'
lib.foo(ALLOCATOR(allocate))
print 'foo returned'

print allocated_arrays[0]
```

The allocate function creates a new !NumPy array and puts it in a list so that we keep a reference to it after the callback function returns. Expected output:

```
calling foo
foo calling allocator
allocate called
0xaf5778
allocate returning
allocator returned in foo
data = 0x00AF5778
foo returned
[[ 1\.  2\.  3.]
 [ 2\.  4\.  6.]]
```

Here’s another idea for an Allocator class to manage this kind of thing. In addition to dimension and shape, this allocator function takes a char indicating what type of array to allocate. You can get these typecodes from the ndarrayobject.h header, in the `NPY_TYPECHAR` enum.

```
from ctypes import *
import numpy as N

class Allocator:
    CFUNCTYPE = CFUNCTYPE(c_long, c_int, POINTER(c_int), c_char)

    def __init__(self):
        self.allocated_arrays = []

    def __call__(self, dims, shape, dtype):
        x = N.empty(shape[:dims], N.dtype(dtype))
        self.allocated_arrays.append(x)
        return x.ctypes.data_as(c_void_p).value

    def getcfunc(self):
        return self.CFUNCTYPE(self)
    cfunc = property(getcfunc)
```

Use it like this in Python:

```
lib.func.argtypes = [..., Allocator.CFUNCTYPE]
def func():
    alloc = Allocator()
    lib.func(..., alloc.cfunc)
    return tuple(alloc.allocated_arrays[:3])
```

Corresponding C code:

```
typedef void*(*allocator_t)(int, int*, char);

void func(..., allocator_t allocator) {
   /* ... */
   int dims[] = {2, 3, 4};
   double* data = (double*) allocator(3, dims, 'd');
   /* allocate more arrays here */
}
```

None of the allocators presented above are thread safe. If you have multiple Python threads calling the C code that invokes your callbacks, you will have to do something a bit smarter.

More useful code frags

Suppose you have a C function like the following, which operates on a pointer-to-pointers data structure.

```
void foo(float** data, int len) {
    float** x = data;
    for (int i = 0; i < len; i++, x++) {
        /* do something with *x */
    }
}
```

You can create the necessary structure from an existing 2-D !NumPy array using the following code:

```
x = N.array([[10,20,30], [40,50,60], [80,90,100]], 'f4')
f4ptr = POINTER(c_float)
data = (f4ptr*len(x))(*[row.ctypes.data_as(f4ptr) for row in x])
```

`f4ptr*len(x)` creates a ctypes array type that is just large enough to contain a pointer to every row of the array.

Heterogeneous Types Example

Here's a simple example when using heterogeneous dtypes (record arrays).

But, be warned that NumPy recarrays and corresponding structs in C ***may not*** be congruent.

Also structs are not standardized across platforms ...In other words, **'' be aware of padding issues!**''

sample.c

```
#include <stdio.h>

typedef struct Weather_t {
    int timestamp;
    char desc[12];
} Weather;

void print_weather(Weather* w, int nelems)
{
    int i;
    for (i=0;i<nelems;++i) {
        printf("timestamp: %d\ndescription: %s\n\n", w[i].timestamp
    }
}
```

SConstruct

```
env = Environment()
env.Replace(CFLAGS=['-O2','-Wall','-ansi','-pedantic'])
env.SharedLibrary('sample', ['sample.c'])
```

sample.py

```
import numpy as N
import ctypes as C

dat = [[1126877361,'sunny'], [1126877371,'rain'], [1126877385,'damn

dat_dtype = N.dtype([('timestamp','i4'),('desc','|S12')])
arr = N.rec.fromrecords(dat,dtype=dat_dtype)

_sample = N.ctypeslib.load_library('libsample','.')
_sample.print_weather.restype = None
_sample.print_weather.argtypes = [N.ctypeslib.ndpointer(dat_dtype,

def print_weather(x):
    _sample.print_weather(x, x.size)

if __name__=='__main__':
    print_weather(arr)
```

Fibonacci example (using NumPy arrays, C and Scons)

The following was tested and works on Windows (using MinGW) and GNU/Linux 32-bit OSs (last tested 13-08-2009). Copy all three files to the same directory.

The C code (this calculates the Fibonacci number recursively):

```
/*
    Filename: fibonacci.c
    To be used with fibonacci.py, as an imported library. Use Scons
    simply type 'scons' in the same directory as this file (see www
*/

/* Function prototypes */
int fib(int a);
void fibseries(int *a, int elements, int *series);
void fibmatrix(int *a, int rows, int columns, int *matrix);

int fib(int a)
{
    if (a <= 0) /*  Error -- wrong input will return -1\. */
        return -1;
    else if (a==1)
        return 0;
    else if ((a==2)||(a==3))
        return 1;
    else
        return fib(a - 2) + fib(a - 1);
}

void fibseries(int *a, int elements, int *series)
{
    int i;
    for (i=0; i < elements; i++)
    {
    series[i] = fib(a[i]);
    }
}

void fibmatrix(int *a, int rows, int columns, int *matrix)
{
    int i, j;
    for (i=0; i<rows; i++)
        for (j=0; j<columns; j++)
        {
            matrix[i * columns + j] = fib(a[i * columns + j]);
        }
}
```

The Python code:

```
"""
Filename: fibonacci.py
Demonstrates the use of ctypes with three functions:

 (1) fib(a)
 (2) fibseries(b)
```

```
 (3) fibmatrix(c)
"""

import numpy as nm
import ctypes as ct

# Load the library as _libfibonacci.
# Why the underscore (_) in front of _libfibonacci below?
# To mimimise namespace pollution -- see PEP 8 (www.python.org).
_libfibonacci = nm.ctypeslib.load_library('libfibonacci', '.')

_libfibonacci.fib.argtypes = [ct.c_int] #  Declare arg type, same b
_libfibonacci.fib.restype  =  ct.c_int  #  Declare result type, sar

_libfibonacci.fibseries.argtypes = [nm.ctypeslib.ndpointer(dtype =
                                    ct.c_int,\
                                    nm.ctypeslib.ndpointer(dtype =
_libfibonacci.fibseries.restype  = ct.c_void_p

_libfibonacci.fibmatrix.argtypes = [nm.ctypeslib.ndpointer(dtype =
                                    ct.c_int, ct.c_int,\
                                   nm.ctypeslib.ndpointer(dtype =
_libfibonacci.fibmatrix.restype  = ct.c_void_p

def fib(a):
    """Compute the n'th Fibonacci number.

 ARGUMENT(S):
 An integer.

 RESULT(S):
 The n'th Fibonacci number.

 EXAMPLE(S):
 >>> fib(8)
 13
 >>> fib(23)
 17711
 >>> fib(0)
 -1
 """
    return _libfibonacci.fib(int(a))

def fibseries(b):
    """Compute an array containing the n'th Fibonacci number of eac

 ARGUMENT(S):
 A list or NumPy array (dim = 1) of integers.

 RESULT(S):
 NumPy array containing the n'th Fibonacci number of each entry.

 EXAMPLE(S):
```

```
 >>> fibseries([1,2,3,4,5,6,7,8])
 array([ 0,  1,  1,  2,  3,  5,  8, 13])
 >>> fibseries(range(1,12))
 array([ 0,  1,  1,  2,  3,  5,  8, 13, 21, 34, 55])
 """
    b = nm.asarray(b, dtype=nm.intc)
    result = nm.empty(len(b), dtype=nm.intc)
    _libfibonacci.fibseries(b, len(b), result)
    return result

def fibmatrix(c):
    """Compute a matrix containing the n'th Fibonacci number of eac

 ARGUMENT(S):
 A nested list or NumPy array (dim = 2) of integers.

 RESULT(S):
 NumPy array containing the n'th Fibonacci number of each entry.

 EXAMPLE(S):
 >>> from numpy import array
 >>> fibmatrix([[3,4],[5,6]])
 array([[1, 2],
 [3, 5]])
 >>> fibmatrix(array([[1,2,3],[4,5,6],[7,8,9]]))
 array([[ 0,  1,  1],
 [ 2,  3,  5],
 [ 8, 13, 21]])
 """
    tmp = nm.asarray(c)
    rows, cols = tmp.shape
    c = tmp.astype(nm.intc)
    result = nm.empty(c.shape, dtype=nm.intc)
    _libfibonacci.fibmatrix(c, rows, cols, result)
    return result
```

Here’s the SConstruct file contents (filename: SConstruct):

```
env = Environment()
env.Replace(CFLAGS=['-O2', '-Wall', '-ansi', '-pedantic'])
env.SharedLibrary('libfibonacci', ['fibonacci.c'])
```

In Python interpreter (or whatever you use), do:

```
>>> import fibonacci as fb
>>> fb.fib(8)
13
>>> fb.fibseries([5,13,2,6]
array([  3, 144,    1,    5])
```

Pertinent Mailing List Threads

Some useful threads on the ctypes-users mailing list:

- [http://aspn.activestate.com/ASPN/Mail/Message/ctypes-users/3119087 IndexError when indexing on POINTER(POINTER(ctype))]
- [http://aspn.activestate.com/ASPN/Mail/Message/ctypes-users/3118513 Adding ctypes support to NumPy]
- [http://aspn.activestate.com/ASPN/Mail/Message/ctypes-users/3118656 Determining if a ctype is a pointer type (was RE: Adding ctypes support to NumPy)]
- [http://aspn.activestate.com/ASPN/Mail/Message/ctypes-users/3117306 Check for NULL pointer without ValueError]
- [http://aspn.activestate.com/ASPN/Mail/Message/ctypes-users/3205951 Problem with callbacks from C into Python]
- [http://thread.gmane.org/gmane.comp.python.numeric.general/7418 ctypes and ndpointer]
- [http://thread.gmane.org/gmane.comp.python.ctypes/3116 Problems with 64 signed integer]

Thomas Heller's answers are particularly insightful.

Documentation

- [http://starship.python.net/crew/theller/ctypes/tutorial.html ctypes tutorial]
- [http://docs.python.org/dev/lib/module-ctypes.html 13.14 ctypes – A foreign function library for Python.]

F2py

This page provides examples on how to use the F2py fortran wrapping program.

It is possible to start with simple routines or even to wrap full Fortran modules. F2py is used in SciPy itself and you can find some examples in the source code of SciPy.

Short examples

Wrapping a function from lapack

Taken from a message on 2006-06-22 to scipy-user by ArndBaecker

Thanks to f2py, wrapping Fortran code is (with a bit of effort) trivial in many cases. For complicated functions requiring many arguments the wrapper can become longish. Fortunately, many things can be learnt from looking at `scipy/Lib/linalg/generic_flapack.pyf` In particular, the documentation at http://cens.ioc.ee/projects/f2py2e/ is excellent. I also found the f2py notes by FernandoPerez very helpful, http://cens.ioc.ee/pipermail/f2py-users/2003-April/000472.html

Let me try to give some general remarks on how to start (the real authority on all this is of course Pearu, so please correct me if I got things wrong here): *first find a routine which will do the job you want:* If the lapack documentation is installed properly on Linux you could do *http://www.netlib.org/ provides a nice decision tree* make sure that that it does not exist in scipy:

```
from scipy.lib import lapack
lapack.clapack.<TAB>         (assuming Ipython)
lapack.clapack.<routine_name>
```

```
Remark: routines starting with c/z are for double/single complex`` a
```

```
* You can use this by
```

```
import wrap_lap
```

```
Note, that this is not yet polished (this is the part on`` which ha
```

Concrete (and very simple) example (non-lapack):

Wrapping Hermite polynomials

Download code (found after hours of googling ;-) , from
http://cdm.unimo.it/home/matematica/funaro.daniele/splib.txt

and extract

Generate wrapper framework:

```
# only run the following line _once_
# (and never again, otherwise the hand-modified hermite.pyf
#  goes down the drains)
f2py -m hermite -h hermite.pyf hermite.f
```

Then modify

Create the module:

```
f2py -c hermite.pyf  hermite.f

# add this if you want:
-DF2PY_REPORT_ON_ARRAY_COPY=1 -DF2PY_REPORT_ATEXIT
```

Simple test:

```
import hermite
hermite.vahepo(2,2.0)
import scipy
scipy.special.hermite(2)(2.0)
```

A more complicated example about how to wrap routines for band matrices can be found at http://www.physik.tu-dresden.de/~baecker/comp_talks.html under “Python and Co - some recent developments”.

Wrapping a simple C code

f2py is also capable of handling C code. An example is found on the wiki: [“Cookbook/f2py_and_NumPy”].

Step by step wrapping of a simple numerical code: Interactive System for Ice sheet Simulation

http://websrv.cs.umt.edu/isis/index.php/F2py_example

Inline Weave With Basic Array Conversion (no Blitz)

Python and Numpy are designed to express general statements that work transparently on many sizes of incoming data. Using inline Weave with Blitz conversion can dramatically speed up many numerical operations (eg, addition of a series of arrays) because in some ways it bypasses generality. How can you speed up your algorithms with inline C code while maintaining generality? One tool provided by Numpy is the **iterator**. Because an iterator keeps track of memory indexing for you, its operation is very analogous to the concept of iteration in Python itself. You can write loop in C that simply says “take the next element in a serial object–the !PyArrayObject–and operate on it, until there are no more elements.”

This is a very simple example of multi dimensional iterators, and their power to “broadcast” arrays of compatible shapes. It shows that the very same code that is entirely ignorant of dimensionality can achieve completely different computations based on the rules of broadcasting. I have assumed in this case that ***a*** has at least as many dimensions as ***b***. It is important to know that the weave array conversion of ***a*** gives you access in C++ to: *py_a – !PyObject a_array – !PyArrayObject a – (c-type*) py_array->data

```
import numpy as npy
from scipy.weave import inline

def multi_iter_example():
    a = npy.ones((4,4), npy.float64)
    # for the sake of driving home the "dynamic code" approach...
    dtype2ctype = {
        npy.dtype(npy.float64): 'double',
        npy.dtype(npy.float32): 'float',
        npy.dtype(npy.int32): 'int',
        npy.dtype(npy.int16): 'short',
    }
    dt = dtype2ctype.get(a.dtype)

    # this code does a = a*b inplace, broadcasting b to fit the sha
    code = \
"""
%s *p1, *p2;
PyObject *itr;
itr = PyArray_MultiIterNew(2, a_array, b_array);
while(PyArray_MultiIter_NOTDONE(itr)) {
 p1 = (%s *) PyArray_MultiIter_DATA(itr, 0);
 p2 = (%s *) PyArray_MultiIter_DATA(itr, 1);
 *p1 = (*p1) * (*p2);
 PyArray_MultiIter_NEXT(itr);
}
""" % (dt, dt, dt)

    b = npy.arange(4, dtype=a.dtype)
    print '\n A                  B     '
    print a, b
    # this reshaping is redundant, it would be the default broadcas
    b.shape = (1,4)
    inline(code, ['a', 'b'])
    print "\ninline version of a*b[None,:],"
    print a
    a = npy.ones((4,4), npy.float64)
    b = npy.arange(4, dtype=a.dtype)
    b.shape = (4,1)
    inline(code, ['a', 'b'])
    print "\ninline version of a*b[:,None],"
    print a
```

There are two other iterator applications in `iterators_example.py` and `iterators.py` .

Deeper into the “inline” method

The docstring for **inline** is enormous, and indicates that all kinds of compiling options are supported when integrating your inline code. I've taken advantage of this to make some specialized FFTW calls a lot more simple, and in only a few additional lines add support for inline FFTs. In this example, I read in a file of pure C code and use it as ***support_code*** in my inline statement. I also use a tool from Numpy's distutils to locate my FFTW libraries and headers.

```
import numpy as N
from scipy.weave import inline
from os.path import join, split
from numpy.distutils.system_info import get_info

fft1_code = \
"""
char *i, *o;
i = (char *) a;
o = inplace ? i : (char *) b;
if(isfloat) {
 cfft1d(reinterpret_cast<fftwf_complex*>(i),
 reinterpret_cast<fftwf_complex*>(o),
 xdim, len_array, direction, shift);
} else {
 zfft1d(reinterpret_cast<fftw_complex*>(i),
 reinterpret_cast<fftw_complex*>(o),
 xdim, len_array, direction, shift);
}
"""
extra_code = open(join(split(__file__)[0],'src/cmplx_fft.c')).read(
fftw_info = get_info('fftw3')

def fft1(a, shift=True, inplace=False):
    if inplace:
        _fft1_work(a, -1, shift, inplace)
    else:
        return _fft1_work(a, -1, shift, inplace)

def ifft1(a, shift=True, inplace=False):
    if inplace:
        _fft1_work(a, +1, shift, inplace)
    else:
        return _fft1_work(a, +1, shift, inplace)

def _fft1_work(a, direction, shift, inplace):
    # to get correct C-code, b always must be an array (but if it's
    # not being used, it can be trivially small)
    b = N.empty_like(a) if not inplace else N.array([1j], a.dtype)
    inplace = 1 if inplace else 0
    shift = 1 if shift else 0
    isfloat = 1 if a.dtype.itemsize==8 else 0
    len_array = N.product(a.shape)
    xdim = a.shape[-1]
    inline(fft1_code, ['a', 'b', 'isfloat', 'inplace',
```

```
                        'len_array', 'xdim', 'direction', 'shift'],
            support_code=extra_code,
            headers=['<fftw3.h>'],
            libraries=['fftw3', 'fftw3f'],
            include_dirs=fftw_info['include_dirs'],
            library_dirs=fftw_info['library_dirs'],
            compiler='gcc')
    if not inplace:
        return b
```

This code is available in [attachment:fftmod.tar.gz fftmod.tar.gz].

Attachments

- `fftmod.tar.gz`
- `iterators.py`
- `iterators_example.py`

Pyrex And NumPy

Please note that the code described here is slightly out of date, since today cython is the actively maintained version of Pyrex, and numpy now ships with Cython examples.

Rather than maintaining both the wiki and the source dir, we'll continue to update the sources, kept here.

Old Pyrex page

Pyrex is a language for writing C extensions to Python. Its syntax is very similar to writing Python. A file is compiled to a file, which is then compiled like a standard C extension module for Python. Many people find writing extension modules with Pyrex preferable to writing them in C or using other tools, such as SWIG.

This page is a starting point for accessing numpy arrays natively with Pyrex. Please note that with current versions of NumPy (SVN), the directory contains a complete working example with the code in this page, including also a proper file so you can install it with the standard Python mechanisms. This should help you get up and running quickly.

Here's a file I call "c_python.pxd":

```
cdef extern from "Python.h":
    ctypedef int Py_intptr_t
```

and here's "c_numpy.pxd":

```
cimport c_python

cdef extern from "numpy/arrayobject.h":
    ctypedef class numpy.ndarray [object PyArrayObject]:
        cdef char *data
        cdef int nd
        cdef c_python.Py_intptr_t *dimensions
        cdef c_python.Py_intptr_t *strides
        cdef object base
        # descr not implemented yet here...
        cdef int flags
        cdef int itemsize
        cdef object weakreflist

    cdef void import_array()
```

Here’s an example program, name this something like “test.pyx” suffix.

```
cimport c_numpy
cimport c_python
import numpy

c_numpy.import_array()

def print_array_info(c_numpy.ndarray arr):
    cdef int i

    print '-='*10
    print 'printing array info for ndarray at 0x%0lx'%(<c_python.Py
    print 'print number of dimensions:',arr.nd
    print 'address of strides: 0x%0lx'%(<c_python.Py_intptr_t>arr.s
    print 'strides:'
    for i from 0<=i<arr.nd:
        # print each stride
        print '  stride %d:'%i,<c_python.Py_intptr_t>arr.strides[i]
    print 'memory dump:'
    print_elements( arr.data, arr.strides, arr.dimensions, arr.nd,
    print '-='*10
    print

cdef print_elements(char *data,
                    c_python.Py_intptr_t* strides,
                    c_python.Py_intptr_t* dimensions,
                    int nd,
                    int elsize,
                    object dtype):
    cdef c_python.Py_intptr_t i,j
    cdef void* elptr

    if dtype not in [numpy.dtype(numpy.object_),
                     numpy.dtype(numpy.float64)]:
        print '   print_elements() not (yet) implemented for dtype
        return

    if nd ==0:
        if dtype==numpy.dtype(numpy.object_):
            elptr = (<void**>data)[0] #[0] dereferences pointer in
            print '  ',<object>elptr
        elif dtype==numpy.dtype(numpy.float64):
            print '  ',(<double*>data)[0]
    elif nd == 1:
        for i from 0<=i<dimensions[0]:
            if dtype==numpy.dtype(numpy.object_):
                elptr = (<void**>data)[0]
                print '  ',<object>elptr
            elif dtype==numpy.dtype(numpy.float64):
                print '  ',(<double*>data)[0]
            data = data + strides[0]
```

```
    else:
        for i from 0<=i<dimensions[0]:
            print_elements(data, strides+1, dimensions+1, nd-1, els
            data = data + strides[0]

def test():
    """this function is pure Python"""
    arr1 = numpy.array(-1e-30,dtype=numpy.Float64)
    arr2 = numpy.array([1.0,2.0,3.0],dtype=numpy.Float64)

    arr3 = numpy.arange(9,dtype=numpy.Float64)
    arr3.shape = 3,3

    four = 4
    arr4 = numpy.array(['one','two',3,four],dtype=numpy.object_)

    arr5 = numpy.array([1,2,3]) # int types not (yet) supported by

    for arr in [arr1,arr2,arr3,arr4,arr5]:
        print_array_info(arr)
```

Now, if you compile and install the above test.pyx, the output of should be something like the following:

```
-=-=-=-=-=-=-=-=-=-=
printing array info for ndarray at 0x8184508
print number of dimensions: 0
address of strides: 0xb764f7ec
strides:
memory dump:
   -1e-30
-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=
printing array info for ndarray at 0x8190060
print number of dimensions: 1
address of strides: 0x818453c
strides:
  stride 0: 8
memory dump:
   1.0
   2.0
   3.0
-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=
printing array info for ndarray at 0x82698a0
print number of dimensions: 2
address of strides: 0x8190098
strides:
```

```
  stride 0: 24
  stride 1: 8
memory dump:
   0.0
   1.0
   2.0
   3.0
   4.0
   5.0
   6.0
   7.0
   8.0
-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=
printing array info for ndarray at 0x821d6e0
print number of dimensions: 1
address of strides: 0x818ed74
strides:
  stride 0: 4
memory dump:
   one
   two
   3
   4
-=-=-=-=-=-=-=-=-=-=

-=-=-=-=-=-=-=-=-=-=
printing array info for ndarray at 0x821d728
print number of dimensions: 1
address of strides: 0x821d75c
strides:
  stride 0: 4
memory dump:
   print_elements() not (yet) implemented for dtype int32
-=-=-=-=-=-=-=-=-=-=
```

The [http://pytables.sourceforge.net/ pytables project] makes extensive use of Pyrex and numarray. See the pytables source code for more ideas.

= See Also = [“Cookbook/ArrayStruct_and_Pyrex”]

SWIG Numpy examples

Introduction

These are simple !NumPy and SWIG examples which use the numpy.i interface file. There is also a MinGW section for people who may want to use these in a Win32 environment. The following code is C, rather than C++.

The information contained here was first made available by Bill Spotz in his article *numpy.i: a SWIG Interface File for !NumPy*, and the !NumPy SVN which can be checked out using the following command:

```
svn co http://scipy.org/svn/numpy/trunk numpy
```

`* The !NumPy+SWIG manual is available here: ```1` [http://scipy.org/svn/numpy/trunk/doc/swig/doc/numpy_swig.pdf] (http://scipy.org/svn/numpy/trunk/doc/swig/doc/numpy_swig.pdf) `__` *The numpy.i file can be downloaded from the SVN:* `` ``2 `` *[http://scipy.org/svn/numpy/trunk/doc/swig/numpy.i] (http://scipy.org/svn/numpy/trunk/doc/swig/numpy.i)* `__` and the pyfragments.swg file, hich is also needed, is available from
`3` <[http://scipy.org/svn/numpy/trunk/doc/swig/pyfragments.swg](htt`
4 `<[http://sourceforge.net/project/showfiles.php?group_id=1369&packa`

__

Initial setup

gcc and SWIG

Check that both gcc and SWIG are available (paths known):

```
swig -version
```

and

```
gcc -v
```

Both should output some text...

Modifying the pyfragments.swg file (MinGW only)

This is from my own tests, running SWIG Version 1.3.36 and gcc version 3.4.5 (mingw-vista special r3). I had to remove the 'static' statements from the source, otherwise your SWIGed sources won't compile. There are only two 'static' statements in the file, both will need removing. Here is my modified version: pyfragments.swg

Compilation and testing

A setup.py file specific to each module must be written first. I based mine on the reference setup.py available in http://scipy.org/svn/numpy/trunk/doc/swig/test/ with added automatic handling of swig.

On a un*x like system, the command-line is:

```
python setup.py build
```

In a Win32 environment (either cygwin or cmd), the setup command-line is (for use with MinGW):

```
python setup.py build --compiler=mingw32
```

The command handles both the SWIG process (generation of wrapper C and Python code) and gcc compilation. The resulting module (a pyd file) is built in the `build\lib.XXX` directory (e.g. for a Python 2.5 install and on a Win32 machine, the `build\lib.win32-2.5` directory).

A simple ARGOUT_ARRAY1 example

This is a re-implementation of the range function. The module is called ezrange. One thing to remember with `ARGOUT_ARRAY1` is that the dimension of the array must be passed from Python.

From Bill Spotz's article: *The python user does not pass these arrays in, they simply get returned. For the case where a dimension is specified, the python user must provide that dimension as an argument.*

This is useful for functions like `numpy.arange(N)`, for which the size of the returned array is known in advance and passed to the C function.

For functions that follow `array_out = function(array_in)` where the size of array*out is _not* known in advance and depends on memory allocated in C, see the example given in [:Cookbook/SWIG Memory Deallocation].

The C source (ezrange.c and ezrange.h)

Here is the ezrange.h file:

```
void range(int *rangevec, int n);
```

Here is the ezrange.c file:

```
void range(int *rangevec, int n)
{
    int i;

    for (i=0; i< n; i++)
        rangevec[i] = i;
}
```

The interface file (ezrange.i)

Here is the ezrange.i file.

```
%module ezrange

%{
    #define SWIG_FILE_WITH_INIT
    #include "ezrange.h"
%}

%include "numpy.i"

%init %{
    import_array();
%}

%apply (int* ARGOUT_ARRAY1, int DIM1) {(int* rangevec, int n)}

%include "ezrange.h"
```

Don’t forget that you will also need the numpy.i file in the same directory.

Setup file (setup.py)

This is my setup.py file:

```
#! /usr/bin/env python

# System imports
from distutils.core import *
from distutils      import sysconfig

# Third-party modules - we depend on numpy for everything
import numpy

# Obtain the numpy include directory.  This logic works across nump
try:
    numpy_include = numpy.get_include()
except AttributeError:
    numpy_include = numpy.get_numpy_include()

# ezrange extension module
_ezrange = Extension("_ezrange",
                   ["ezrange.i","ezrange.c"],
                   include_dirs = [numpy_include],
                   )

# ezrange setup
setup(  name        = "range function",
        description = "range takes an integer and returns an n elem
        author      = "Egor Zindy",
        version     = "1.0",
        ext_modules = [_ezrange]
        )
```

Compiling the module

The setup command-line is:

```
python setup.py build
```

or

```
python setup.py build --compiler=mingw32
```

depending on your environment.

Testing the module

If everything goes according to plan, there should be a `_ezrange.pyd` file available in the `build\lib.XXX` directory. You will need to copy the file in the directory where the `ezrange.py` file is (generated by swig), in which case, the following will work (in python):

```
>>> import ezrange
>>> ezrange.range(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

A simple INPLACE_ARRAY1 example

This example doubles the elements of the 1-D array passed to it. The operation is done in-place, which means that the array passed to the function is changed.

The C source (inplace.c and inplace.h)

Here is the inplace.h file:

```
void inplace(double *invec, int n);
```

Here is the inplace.c file:

```
void inplace(double *invec, int n)
{
    int i;

    for (i=0; i<n; i++)
    {
        invec[i] = 2*invec[i];
    }
}
```

The interface file (inplace.i)

Here is the inplace.i interface file:

```
%module inplace

%{
    #define SWIG_FILE_WITH_INIT
    #include "inplace.h"
%}

%include "numpy.i"

%init %{
    import_array();
%}

%apply (double* INPLACE_ARRAY1, int DIM1) {(double* invec, int n)}
%include "inplace.h"
```

Setup file (setup.py)

This is my setup.py file:

```
#! /usr/bin/env python

# System imports
from distutils.core import *
from distutils      import sysconfig

# Third-party modules - we depend on numpy for everything
import numpy

# Obtain the numpy include directory.  This logic works across nump
try:
    numpy_include = numpy.get_include()
except AttributeError:
    numpy_include = numpy.get_numpy_include()

# inplace extension module
_inplace = Extension("_inplace",
                   ["inplace.i","inplace.c"],
                   include_dirs = [numpy_include],
                   )

# NumyTypemapTests setup
setup(  name        = "inplace function",
        description = "inplace takes a double array and doubles eac

        author      = "Egor Zindy",
        version     = "1.0",
        ext_modules = [_inplace]
        )
```

Compiling the module

The setup command-line is:

```
python setup.py build
```

or

```
python setup.py build --compiler=mingw32
```

depending on your environment.

Testing the module

If everything goes according to plan, there should be a `_inplace.pyd` file available in the `build\lib.XXX` directory. You will need to copy the file in the directory where the `inplace.py` file is (generated by swig), in which case, the following will work (in python):

```
>>> import numpy
>>> import inplace
>>> a = numpy.array([1,2,3],'d')
>>> inplace.inplace(a)
>>> a
array([2., 4., 6.])
```

A simple ARGOUTVIEW_ARRAY1 example

Big fat multiple warnings

Please note, Bill Spotz advises against the use of argout_view arrays, unless absolutely necessary:

```
Argoutview arrays are for when your C code provides you with a view
```

Python does not take care of memory de-allocation, as stated here by Travis Oliphant: 1

```
The tricky part, however, is memory management. How does the memory
```

Memory deallocation is also difficult to handle automatically as there is no easy way to do module “finalization”. There is a `Py_InitModule()` function, but nothing to handle deletion/destruction/finalization (this will be addressed in Python 3000 as stated in PEP3121. In my example, I use the python module atexit but there must be a better way.

Having said all that, if you have no other choice, here is an example that uses ARGOUTVIEW_ARRAY1. As usual, comments welcome!

The module declares a block of memory and a couple of functions: *ezview.set_ones() sets all the elements (doubles) in the memory block to one and returns a numpy array that is a VIEW of the memory block.* ezview.get_view() simply returns a view of the memory block. * ezview.finalize() takes care of the memory deallocation (this is the weak part of this example).

The C source (ezview.c and ezview.h)

Here is the ezview.h file:

```
void set_ones(double *array, int n);
```

Here is the ezview.c file:

```
#include <stdio.h>
#include <stdlib.h>

#include "ezview.h"

void set_ones(double *array, int n)
{
    int i;

    if (array == NULL)
        return;

    for (i=0;i<n;i++)
        array[i] = 1.;
}
```

The interface file (ezview.i)

Here is the ezview.i interface file:

```
%module ezview

%{
    #define SWIG_FILE_WITH_INIT
    #include "ezview.h"

    double *my_array = NULL;
    int my_n = 10;

    void __call_at_begining()
    {
        printf("__call_at_begining...\n");
        my_array = (double *)malloc(my_n*sizeof(double));
    }

    void __call_at_end(void)
    {
        printf("__call_at_end...\n");
        if (my_array != NULL)
            free(my_array);
    }
%}

%include "numpy.i"

%init %{
    import_array();
    __call_at_begining();
```

```
%}

%apply (double** ARGOUTVIEW_ARRAY1, int *DIM1) {(double** vec, int*

%include "ezview.h"
%rename (set_ones) my_set_ones;

%inline %{
void finalize(void){
    __call_at_end();
}

void get_view(double **vec, int* n) {
    *vec = my_array;
    *n = my_n;
}

void my_set_ones(double **vec, int* n) {
    set_ones(my_array,my_n);
    *vec = my_array;
    *n = my_n;
}
%}
```

Don’t forget that you will also need the numpy.i file in the same directory.

Setup file (setup.py)

This is my setup.py file:

```
#! /usr/bin/env python

# System imports
from distutils.core import *
from distutils      import sysconfig

# Third-party modules - we depend on numpy for everything
import numpy

# Obtain the numpy include directory.  This logic works across nump
try:
    numpy_include = numpy.get_include()
except AttributeError:
    numpy_include = numpy.get_numpy_include()

# view extension module
_ezview = Extension("_ezview",
                   ["ezview.i","ezview.c"],
                   include_dirs = [numpy_include],
                   )

# NumyTypemapTests setup
setup(  name        = "ezview module",
        description = "ezview provides 3 functions: set_ones(), get
        author      = "Egor Zindy",
        version     = "1.0",
        ext_modules = [_ezview]
        )
```

Compiling the module

The setup command-line is:

```
python setup.py build
```

or

```
python setup.py build --compiler=mingw32
```

depending on your environment.

Testing the module

If everything goes according to plan, there should be a `_ezview.pyd` file available in the `build\lib.XXX` directory. You will need to copy the file in the directory where the `ezview.py` file is (generated by swig), in which case, the following will work (in python):

The test code test_ezview.py follows:

```
import atexit
import numpy
print "first message is from __call_at_begining()"
import ezview

#There is no easy way to finalize the module (see PEP3121)
atexit.register(ezview.finalize)

a = ezview.set_ones()
print "\ncalling ezview.set_ones() - now the memory block is all on
print a

print "\nwe're setting the array using a[:]=arange(a.shape[0])\nThi
a[:] = numpy.arange(a.shape[0])
print a

print "\nwe're now deleting the array  - this only deletes the view
del a

print "\nlet's get a new view on the allocated memory, should STILL
b = ezview.get_view()
print b

print "\nnext message from __call_at_end() - finalize() registered
```

Launch test_ezview.py and the following will hopefully happen:

```
~> python test_ezview.py
first message is from __call_at_begining()
__call_at_begining...

calling ezview.set_ones() - now the memory block is all ones.
Returned array (a view on the allocated memory block) is:
[ 1\.  1\.  1\.  1\.  1\.  1\.  1\.  1\.  1\.  1.]

we re setting the array using a[:]=arange(a.shape[0])
This changes the content of the allocated memory block:
[ 0\.  1\.  2\.  3\.  4\.  5\.  6\.  7\.  8\.  9.]

we re now deleting the array  - this only deletes the view,
not the allocated memory!

let s get a new view on the allocated memory, should STILL contain
[ 0\.  1\.  2\.  3\.  4\.  5\.  6\.  7\.  8\.  9.]

next message from __call_at_end() - finalize() registered via modul
__call_at_end...
```

Error handling using errno and python exceptions

I have been testing this for a few months now and this is the best I've come-up with. If anyone knows of a better way, please let me know.

From the opengroup website, the lvalue errno is used by many functions to return error values. The idea is that the global variable errno is set to 0 before a function is called (in swig parlance: $action), and checked afterwards. If errno is non-zero, a python exception with a meaningful message is generated, depending on the value of errno.

The following example comprises two examples: First example uses errno when checking whether an array index is valid. Second example uses errno to notify the user of a malloc() problem.

The C source (ezerr.c and ezerr.h)

Here is the ezerr.h file:

```
int val(int *array, int n, int index);
void alloc(int n);
```

Here is the ezerr.c file:

```
#include <stdlib.h>
#include <errno.h>

#include "ezerr.h"

//return the array element defined by index
int val(int *array, int n, int index)
{
    int value=0;

    if (index < 0 || index >=n)
    {
        errno = EPERM;
        goto end;
    }

    value = array[index];

end:
    return value;
}

//allocate (and free) a char array of size n
void alloc(int n)
{
    char *array;

    array = (char *)malloc(n*sizeof(char));
    if (array == NULL)
    {
        errno = ENOMEM;
        goto end;
    }

    //don't keep the memory allocated...
    free(array);

end:
    return;
}
```

The interface file (ezerr.i)

Here is the ezerr.i interface file:

```
%module ezerr
%{
#include <errno.h>
#include "ezerr.h"

#define SWIG_FILE_WITH_INIT
%}

%include "numpy.i"

%init %{
    import_array();
%}

%exception
{
    errno = 0;
    $action

    if (errno != 0)
    {
        switch(errno)
        {
            case EPERM:
                PyErr_Format(PyExc_IndexError, "Index out of range'
                break;
            case ENOMEM:
                PyErr_Format(PyExc_MemoryError, "Failed malloc()")
                break;
            default:
                PyErr_Format(PyExc_Exception, "Unknown exception")
        }
        SWIG_fail;
    }
}

%apply (int* IN_ARRAY1, int DIM1) {(int *array, int n)}

%include "ezerr.h"
```

Note the *SWIG_fail*, which is a macro for *goto fail* in case there is any other cleanup code to execute (thanks Bill!).

Don’t forget that you will also need the numpy.i file in the same directory.

Setup file (setup.py)

This is my setup.py file:

```
#! /usr/bin/env python

# System imports
from distutils.core import *
from distutils      import sysconfig

# Third-party modules - we depend on numpy for everything
import numpy

# Obtain the numpy include directory.  This logic works across nump
try:
    numpy_include = numpy.get_include()
except AttributeError:
    numpy_include = numpy.get_numpy_include()

# err extension module
ezerr = Extension("_ezerr",
                   ["ezerr.i","ezerr.c"],
                   include_dirs = [numpy_include],

                   extra_compile_args = ["--verbose"]
                   )

# NumyTypemapTests setup
setup(  name        = "err test",
        description = "A simple test to demonstrate the use of errr
        author      = "Egor Zindy",
        version     = "1.0",
        ext_modules = [ezerr]
        )
```

Compiling the module

The setup command-line is:

```
python setup.py build
```

or

```
python setup.py build --compiler=mingw32
```

depending on your environment.

Testing the module

If everything goes according to plan, there should be a `_ezerr.pyd` file available in the `build\lib.XXX` directory. You will need to copy the file in the directory where the `ezerr.py` file is (generated by swig), in which case, the following will work (in python):

The test code test_err.py follows:

```
import traceback,sys
import numpy
import ezerr

print "\n--- testing ezerr.val() ---"
a = numpy.arange(10)
indexes = [5,20,-1]

for i in indexes:
    try:
        value = ezerr.val(a,i)
    except:
        print ">> failed for index=%d" % i
        traceback.print_exc(file=sys.stdout)
    else:
        print "using ezerr.val() a[%d]=%d - should be %d" % (i,valu

print "\n--- testing ezerr.alloc() ---"
amounts = [1,-1] #1 byte, -1 byte

for n in amounts:
    try:
        ezerr.alloc(n)
    except:
        print ">> could not allocate %d bytes" % n
        traceback.print_exc(file=sys.stdout)
    else:
        print "allocated (and deallocated) %d bytes" % n
```

Launch test_err.py and the following will hopefully happen:

```
~> python test_err.py

--- testing ezerr.val() ---
using ezerr.val() a[5]=5 - should be 5
>> failed for index=20
Traceback (most recent call last):
  File "test_err.py", line 11, in <module>
    value = ezerr.val(a,i)
IndexError: Index out of range
>> failed for index=-1
Traceback (most recent call last):
  File "test_err.py", line 11, in <module>
    value = ezerr.val(a,i)
IndexError: Index out of range

--- testing ezerr.alloc() ---
allocated (and deallocated) 1 bytes
>> could not allocate -1 bytes
Traceback (most recent call last):
  File "test_err.py", line 23, in <module>
    ezerr.alloc(n)
MemoryError: Failed malloc()
```

Dot product example (from Bill Spotz's article)

The last example given in Bill Spotz's artice is for a dot product function. Here is a fleshed-out version.

The C source (dot.c and dot.h)

Here is the dot.h file:

```
double dot(int len, double* vec1, double* vec2);
```

Here is the dot.c file:

```
#include <stdio.h>
#include "dot.h"

double dot(int len, double* vec1, double* vec2)
{
    int i;
    double d;

    d = 0;
    for(i=0;i<len;i++)
        d += vec1[i]*vec2[i];

    return d;
}
```

The interface files (dot.i and numpy.i)

Here is the complete dot.i file:

```
%module dot

%{
    #define SWIG_FILE_WITH_INIT
    #include "dot.h"
%}

%include "numpy.i"

%init %{
    import_array();
%}

%apply (int DIM1, double* IN_ARRAY1) {(int len1, double* vec1), (in

%include "dot.h"
%rename (dot) my_dot;

%inline %{
    double my_dot(int len1, double* vec1, int len2, double* vec2) {
    if (len1 != len2) {
        PyErr_Format(PyExc_ValueError, "Arrays of lengths (%d,%d) g
        return 0.0;
    }
    return dot(len1, vec1, vec2);
}
%}
```

Setup file (setup.py)

This is the setup.py file:

```
#! /usr/bin/env python

# System imports
from distutils.core import *
from distutils      import sysconfig

# Third-party modules - we depend on numpy for everything
import numpy

# Obtain the numpy include directory.  This logic works across nump
try:
    numpy_include = numpy.get_include()
except AttributeError:
    numpy_include = numpy.get_numpy_include()

# dot extension module
_dot = Extension("_dot",
                   ["dot.i","dot.c"],
                   include_dirs = [numpy_include],
                   )

# dot setup
setup(  name        = "Dot product",
        description = "Function that performs a dot product (numpy
        author      = "Egor Zindy (based on the setup.py file avai
        version     = "1.0",
        ext_modules = [_dot]
        )
```

Compiling the module

The setup command-line is:

```
python setup.py build
```

or

```
python setup.py build --compiler=mingw32
```

depending on your environment.

Testing

If everything goes according to plan, there should be a `_dot.pyd` file available in the `build\lib.XXX` directory. You will need to copy the file in the directory where the `dot.py` file is (generated by swig), in which case, the following will work (in python):

```
>>> import dot
>>> dot.dot([1,2,3],[1,2,3])
14.0
```

Conclusion

That's all folks (for now)! As usual, comments welcome!

- '''TODO''': Code clean-up and moving the examples over to the !SciPy/!NumPy repository?

Regards, Egor

SWIG and Numpy

Please note that with current versions of NumPy (SVN), the directory contains a complete working example with simple SWIG typemaps, including also a proper file so you can install it with the standard Python mechanisms. This should help you get up and running quickly.

To get the feel how to write a truly minimalist interface, below is a relevant part of the simple SWIG interface file `umfpack.i` (this is for SWIG < version 1.3.29) used to wrap the UMFPACK sparse linear solver libraries. The full interface can be found in the directory in the SciPy SVN repository. If you're using SWIG > version 1.3.29, refer to the file in SciPy SVN repository, which is slightly different.

```
/*!
  Gets PyArrayObject from a PyObject.
*/
PyArrayObject *helper_getCArrayObject( PyObject *input, int type,
                                       int minDim, int maxDim ) {
  PyArrayObject *obj;

  if (PyArray_Check( input )) {
    obj = (PyArrayObject *) input;
    if (!PyArray_ISCARRAY( obj )) {
      PyErr_SetString( PyExc_TypeError, "not a C array" );
      return NULL;
    }
    obj = (PyArrayObject *)
      PyArray_ContiguousFromAny( input, type, minDim, maxDim );
    if (!obj) return NULL;
  } else {
    PyErr_SetString( PyExc_TypeError, "not an array" );
    return NULL;
  }
  return obj;
}
%}

/*!
  Use for arrays as input arguments. Could be also used for changin
  in place.

  @a rtype ... return this C data type
  @a ctype ... C data type of the C function
  @a atype ... PyArray_* suffix
*/
#define ARRAY_IN( rtype, ctype, atype ) \
%typemap( python, in ) (ctype *array) { \
  PyArrayObject *obj; \
  obj = helper_getCArrayObject( $input, PyArray_##atype, 1, 1 ); \
```

```
    if (!obj) return NULL; \
    $1 = (rtype *) obj->data; \
    Py_DECREF( obj ); \
};

ARRAY_IN( int, const int, INT )
%apply const int *array {
    const int Ap [ ],
    const int Ai [ ]
};

ARRAY_IN( long, const long, LONG )
%apply const long *array {
    const long Ap [ ],
    const long Ai [ ]
};

ARRAY_IN( double, const double, DOUBLE )
%apply const double *array {
    const double Ax [ ],
    const double Az [ ],
    const double B [ ]
};

ARRAY_IN( double, double, DOUBLE )
%apply double *array {
    double X [ ]
};
```

The function being wrapped then could be like:

```
int umfpack_di_solve( int sys, const int Ap [ ], const int Ai [ ],
                      const double Ax [ ], double X [ ], const douk
                      ... );
```

Attachments

- `umfpack.i`

SWIG memory deallocation

Introduction

Recipe description

This cookbook recipe describes the automatic deallocation of memory blocks allocated via `malloc()` calls in C, when the corresponding Python numpy array objects are destroyed. The recipe uses SWIG and a modified `numpy.i` helper file.

To be more specific, new fragments were added to the existing `numpy.i` to handle automatic deallocation of arrays, the size of which is not known in advance. As with the original fragments, a block of `malloc()` memory can be converted into a returned numpy python object via a call to `PyArray_SimpleNewFromData()` . However, the returned python object is created using `PyCObject_FromVoidPtr()` , which ensures that the allocated memory is automatically disposed of when the Python object is destroyed. Examples below show how using these new fragments avoids leaking memory.

Since the new fragments are based on the `_ARGOUTVIEW_` ones, the name `_ARGOUTVIEWM_` was chosen, where `M` stands for managed. All managed fragments (ARRAY1, 2 and 3, FARRAY1, 2 and 3) were implemented, and have now been extensively tested.

Where to get the files

At the moment, the modified numpy.i file is available here (last updated 2012-04-22): *http://ezwidgets.googlecode.com/svn/trunk/numpy/numpy.i*
http://ezwidgets.googlecode.com/svn/trunk/numpy/pyfragments.swg

How the code came about

The original memory deallocation code was written by Travis Oliphant (see http://blog.enthought.com/?p=62) and as far as I know, these clever people were the first ones to use it in a swig file (see http://niftilib.sourceforge.net/pynifti, file nifticlib.i). Lisandro Dalcin then pointed out a simplified implementation using CObjects, which Travis details in this updated blog post.

How to use the new fragments

Important steps

In yourfile.i, the %init function uses the same `import_array()` call you already know:

```
%init %{
    import_array();
%}
```

... then just use ARGOUTVIEWM_ARRAY1 instead of ARGOUTVIEW_ARRAY1 and memory deallocation is handled automatically when the python array is destroyed (see examples below).

A simple ARGOUTVIEWM_ARRAY1 example

The SWIG-wrapped function in C creates an N integers array, using `malloc()` to allocate memory. From python, this function is repetitively called and the array created destroyed (M times).

Using the ARGOUTVIEW_ARRAY1 provided in numpy.i, this will create memory leaks (I know ARGOUTVIEW_ARRAY1 has not been designed for this purpose but it's tempting!).

Using the ARGOUTVIEWM_ARRAY1 fragment instead, the memory allocated with `malloc()` will be automatically deallocated when the array is deleted.

The python test program creates and deletes a 1024^2 ints array 2048 times using both ARGOUTVIEW_ARRAY1 and ARGOUTVIEWM_ARRAY1 and when memory allocation fails, an exception is generated in C and caught in Python, showing which iteration finally caused the allocation to fail.

The C source (ezalloc.c and ezalloc.h)

Here is the ezalloc.h file:

```
void alloc(int ni, int** veco, int *n);
```

Here is the ezalloc.c file:

```
#include <stdio.h>
#include <errno.h>
#include "ezalloc.h"

void alloc(int ni, int** veco, int *n)
{
    int *temp;
    temp = (int *)malloc(ni*sizeof(int));

    if (temp == NULL)
        errno = ENOMEM;

    //veco is either NULL or pointing to the allocated block of mem
    *veco = temp;
    *n = ni;
}
```

The interface file (ezalloc.i)

The file (available here: ezalloc.i) does a couple of interesting things: *Like I said in the introduction, calling the* `import_array()` *function in the* `%init` *section now also initialises the memory deallocation code. There is nothing else to add here.* An exception is generated if memory allocation fails. After a few iterations of the code construct, using `errno` and `SWIG_fail` is the simplest I've come-up with. * In this example, two inline functions are created, one using ARGOUTVIEW_ARRAY1 and the other ARGOUTVIEWM_ARRAY1. Both function use the `alloc()` function (see ezalloc.h and ezalloc.c).

```
%module ezalloc
%{
#include <errno.h>
#include "ezalloc.h"

#define SWIG_FILE_WITH_INIT
%}

%include "numpy.i"

%init %{
    import_array();
%}

%apply (int** ARGOUTVIEWM_ARRAY1, int *DIM1) {(int** veco1, int* n1
%apply (int** ARGOUTVIEW_ARRAY1, int *DIM1) {(int** veco2, int* n2)

%include "ezalloc.h"

%exception
```

```
{
    errno = 0;
    $action

    if (errno != 0)
    {
        switch(errno)
        {
            case ENOMEM:
                PyErr_Format(PyExc_MemoryError, "Failed malloc()");
                break;
            default:
                PyErr_Format(PyExc_Exception, "Unknown exception");
        }
        SWIG_fail;
    }
}

%rename (alloc_managed) my_alloc1;
%rename (alloc_leaking) my_alloc2;

%inline %{

void my_alloc1(int ni, int** veco1, int *n1)
{
    /* The function... */
    alloc(ni, veco1, n1);
}

void my_alloc2(int ni, int** veco2, int *n2)
{
    /* The function... */
    alloc(ni, veco2, n2);
}

%}
```

Don’t forget that you will need the numpy.i file in the same directory for this to compile.

Setup file (setup_alloc.py)

This is the setup_alloc.py file:

```
#! /usr/bin/env python

# System imports
from distutils.core import *
from distutils      import sysconfig

# Third-party modules - we depend on numpy for everything
import numpy

# Obtain the numpy include directory.  This logic works across nump
try:
    numpy_include = numpy.get_include()
except AttributeError:
    numpy_include = numpy.get_numpy_include()

# alloc extension module
_ezalloc = Extension("_ezalloc",
                   ["ezalloc.i","ezalloc.c"],
                   include_dirs = [numpy_include],

                   extra_compile_args = ["--verbose"]
                   )

# NumyTypemapTests setup
setup(  name        = "alloc functions",
        description = "Testing managed arrays",
        author      = "Egor Zindy",
        version     = "1.0",
        ext_modules = [_ezalloc]
        )
```

Compiling the module

The setup command-line is (in Windows, using mingw):

```
$> python setup_alloc.py build --compiler=mingw32
```

or in UN*X, simply

```
$> python setup_alloc.py build
```

Testing the module

If everything goes according to plan, there should be a `_ezalloc.pyd` file available in the `build\lib.XXX` directory. The file needs to be copied in the directory with the `ezalloc.py` file (generated by swig).

A python test program is provided in the SVN repository (test_alloc.py) and reproduced below:

```
import ezalloc

n = 2048

# this multiplied by sizeof(int) to get size in bytes...
#assuming sizeof(int)=4 on a 32bit machine (sorry, it's late!)
m = 1024 * 1024
err = 0

print "ARGOUTVIEWM_ARRAY1 (managed arrays) - %d allocations (%d byt
for i in range(n):
    try:
        #allocating some memory
        a = ezalloc.alloc_managed(m)
        #deleting the array
        del a
    except:
        err = 1
        print "Step %d failed" % i
        break

if err == 0:
    print "Done!\n"

print "ARGOUTVIEW_ARRAY1 (unmanaged, leaking) - %d allocations (%d
for i in range(n):
    try:
        #allocating some memory
        a = ezalloc.alloc_leaking(m)
        #deleting the array
        del a
    except:
        err = 1
        print "Step %d failed" % i
        break

if err == 0:
    print "Done? Increase n!\n"
```

Then, a

```
$> python test_alloc.py
```

will produce an output similar to this:

```
ARGOUTVIEWM_ARRAY1 (managed arrays) - 2048 allocations (4194304 byt
Done!

ARGOUTVIEW_ARRAY1 (unmanaged, leaking) - 2048 allocations (4194304
Step 483 failed
```

The unmanaged array leaks memory every time the array view is deleted. The managed one will delete the memory block seamlessly. This was tested both in Windows XP and Linux.

A simple ARGOUTVIEWM_ARRAY2 example

The following examples shows how to return a two-dimensional array from C which also benefits from the automatic memory deallocation.

A naive “crop” function is wrapped using SWIG/numpy.i and returns a slice of the input array. When used as `array_out = crop.crop(array_in, d1_0,d1_1, d2_0,d2_1)`, it is equivalent to the native numpy slicing `array_out = array_in[d1_0:d1_1, d2_0:d2_1]`.

The C source (crop.c and crop.h)

Here is the crop.h file:

```
void crop(int *arr_in, int dim1, int dim2, int d1_0, int d1_1, int
```

Here is the crop.c file:

```
#include <stdlib.h>
#include <errno.h>

#include "crop.h"

void crop(int *arr_in, int dim1, int dim2, int d1_0, int d1_1, int
{
    int *arr=NULL;
    int dim1_o=0;
    int dim2_o=0;
    int i,j;

    //value checks
    if ((d1_1 < d1_0) || (d2_1 < d2_0) ||
```

```
        (d1_0 >= dim1) || (d1_1 >= dim1) || (d1_0 < 0) || (d1_1 < (
        (d2_0 >= dim2) || (d2_1 >= dim2) || (d2_0 < 0) || (d2_1 < (
    {
        errno = EPERM;
        goto end;
    }

    //output sizes
    dim1_o = d1_1-d1_0;
    dim2_o = d2_1-d2_0;

    //memory allocation
    arr = (int *)malloc(dim1_o*dim2_o*sizeof(int));
    if (arr == NULL)
    {
        errno = ENOMEM;
        goto end;
    }

    //copying the cropped arr_in region to arr (naive implementatio
    printf("\n--- d1_0=%d d1_1=%d (rows)  -- d2_0=%d d2_1=%d (colur
    for (j=0; j<dim1_o; j++)
    {
        for (i=0; i<dim2_o; i++)
        {
            arr[j*dim2_o+i] = arr_in[(j+d1_0)*dim2+(i+d2_0)];
            printf("%d ",arr[j*dim2_o+i]);
        }
        printf("\n");
    }
    printf("---\n\n");

end:
    *dim1_out = dim1_o;
    *dim2_out = dim2_o;
    *arr_out = arr;
}
```

The interface file (crop.i)

The file (available here: crop.i) does a couple of interesting things: *The array dimensions DIM1 and DIM2 are in the same order as array.shape on the Python side. In a row major array definition for an image, DIM1 would the number of rows and DIM2 the number of columns.* Using the errno library, An exception is generated when memory allocation fails (ENOMEM) or when a problem occurs with the indexes (EPERM).

```
%module crop
%{
#include <errno.h>
#include "crop.h"

#define SWIG_FILE_WITH_INIT
%}

%include "numpy.i"

%init %{
    import_array();
%}

%exception crop
{
    errno = 0;
    $action

    if (errno != 0)
    {
        switch(errno)
        {
            case EPERM:
                PyErr_Format(PyExc_IndexError, "Index error");
                break;
            case ENOMEM:
                PyErr_Format(PyExc_MemoryError, "Not enough memory'
                break;
            default:
                PyErr_Format(PyExc_Exception, "Unknown exception");
        }
        SWIG_fail;
    }
}

%apply (int* IN_ARRAY2, int DIM1, int DIM2) {(int *arr_in, int dim1
%apply (int** ARGOUTVIEWM_ARRAY2, int* DIM1, int* DIM2) {(int **arr

%include "crop.h"
```

Don’t forget that you will need the numpy.i file in the same directory for this to compile.

Setup file (setup_crop.py)

This is the setup_crop.py file:

```
#! /usr/bin/env python

# System imports
from distutils.core import *
from distutils      import sysconfig

# Third-party modules - we depend on numpy for everything
import numpy

# Obtain the numpy include directory.  This logic works across nump
try:
    numpy_include = numpy.get_include()
except AttributeError:
    numpy_include = numpy.get_numpy_include()

# crop extension module
_crop = Extension("_crop",
                   ["crop.i","crop.c"],
                   include_dirs = [numpy_include],

                   extra_compile_args = ["--verbose"]
                   )

# NumyTypemapTests setup
setup(  name        = "crop test",
        description = "A simple crop test to demonstrate the use of
        author      = "Egor Zindy",
        version     = "1.0",
        ext_modules = [_crop]
        )
```

Testing the module

If everything goes according to plan, there should be a `_crop.pyd` file available in the `build\lib.XXX` directory. The file needs to be copied in the directory with the `crop.py` file (generated by swig).

A python test program is provided in the SVN repository (test_crop.py) and reproduced below:

```
import crop
import numpy

a = numpy.zeros((5,10),numpy.int)
a[numpy.arange(5),:] = numpy.arange(10)

b = numpy.transpose([(10 ** numpy.arange(5))])
a = (a*b)[:,1:] #this array is most likely NOT contiguous

print a
print "dim1=%d dim2=%d" % (a.shape[0],a.shape[1])

d1_0 = 2
d1_1 = 4
d2_0 = 1
d2_1 = 5

c = crop.crop(a, d1_0,d1_1, d2_0,d2_1)
d = a[d1_0:d1_1, d2_0:d2_1]

print "returned array:"
print c

print "native slicing:"
print d
```

This is what the output looks like:

```
$ python test_crop.py
[[    1     2     3     4     5     6     7     8     9]
 [   10    20    30    40    50    60    70    80    90]
 [  100   200   300   400   500   600   700   800   900]
 [ 1000  2000  3000  4000  5000  6000  7000  8000  9000]
 [10000 20000 30000 40000 50000 60000 70000 80000 90000]]
dim1=5 dim2=9

--- d1_0=2 d1_1=4 (rows)  -- d2_0=1 d2_1=5 (columns)
200 300 400 500
2000 3000 4000 5000
---

returned array:
[[ 200  300  400  500]
 [2000 3000 4000 5000]]
native slicing:
[[ 200  300  400  500]
 [2000 3000 4000 5000]]
```

numpy.i takes care of making the array contiguous if needed, so the only thing left to take care of is the array orientation.

Conclusion and comments

That’s all folks! Files are available on the [http://code.google.com/p/ezwidgets/source/browse/#svn/trunk/numpy Google code SVN]. As usual, comments welcome!

Regards, Egor

f2py and numpy

Wrapping C codes using f2py

While initially f2py was developed for wrapping Fortran codes for Python, it can be easily used for wrapping C codes as well. Signature files describing the interface to wrapped functions must be created manually and the functions and their arguments must have the attribute . See f2py UsersGuide for more information about the syntax of signature files.

Here follows as simple C code

```
/* File foo.c */
void foo(int n, double *x, double *y) {
  int i;
  for (i=0;i<n;i++) {
    y[i] = x[i] + i;
  }
}
```

and the corresponding signature file

```
! File m.pyf
python module m
interface
  subroutine foo(n,x,y)
    intent(c) foo                 ! foo is a C function
    intent(c)                     ! all foo arguments are
                                  ! considered as C based
    integer intent(hide), depend(x) :: n=len(x)  ! n is the length
                                                 ! of input array x
    double precision intent(in) :: x(n)          ! x is input array
                                                 ! (or  arbitrary s
    double precision intent(out) :: y(n)         ! y is output arra
                                                 ! see code in foo.
  end subroutine foo
end interface
end python module m
```

To build the wrapper, one can either create a setup.py script

```
# File setup.py
def configuration(parent_package='',top_path=None):
    from numpy.distutils.misc_util import Configuration
    config = Configuration('',parent_package,top_path)

    config.add_extension('m',
                         sources = ['m.pyf','foo.c'])
    return config
if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(**configuration(top_path='').todict())
```

and execute:

```
python setup.py build_src build_ext --inplace
```

Or one can call f2py directly in command line to build the wrapper as follows:

```
f2py m.pyf foo.c -c
```

In both cases an extension module will be created to current directory that can be imported to python:

```
>>> import m
>>> print m.__doc__
This module 'm' is auto-generated with f2py (version:2_2130).
Functions:
 y = foo(x)
.
>>> print m.foo.__doc__
foo - Function signature:
 y = foo(x)
Required arguments:
 x : input rank-1 array('d') with bounds (n)
Return objects:
 y : rank-1 array('d') with bounds (n)

>>> print m.foo([1,2,3,4,5])
[ 1\.  3\.  5\.  7\.  9.]
>>>
```

Outdated

- A numerical agnostic pyrex class
- Array, struct, and Pyrex
- Data Frames
- Python Imaging Library
- Recipes for timeseries
- FAQ
- The FortranFile class
- dbase
- xplt

A numerical agnostic pyrex class

NOTE: This entry was last updated 2006-11-04 and contains information that is not relevant today (as of 2013).

Overview

Here is presented *!NumInd* (Numerical Independent Extension), an example of a class written in Pyrex. This class can wrap information from any object coming from Numeric, numarray or !NumPy without any dependency of those packages for compiling the extension. It lets you to create a uniform interface in both Python and C spaces. In addition, you can personalize it by adding new methods or properties.

For this extension to work, you need a numerical package that supports the array interface. Any of these versions would be good enough:

* NumPy (all versions) * Numeric (>=24.2) * numarray (>=1.5.1)

NumInd: a Numerical Independent Pyrex-based extension

The !NumInd class shown below takes a Numeric/numarray/!NumPy object and creates another object that can be accessed in an uniform way from both Python and Pyrex (and hence, C) space. Moreover, it exposes an array interface so that you can re-wrap this object with any Numeric/numarray/!NumPy. All of these features are achieved without actually copying the data itself. This opens the door to the possibility to develop applications that supports the Numeric/numarray/!NumPy triad without a need to compile against any of them.

Warning: This class supports mainly homogeneous datasets, but it wouldn't be difficult to support recarrays as well. This is a work in-progress anyway.

```
# This Pyrex extension class can take a numpy/numarray/Numeric obje
# as a parameter and wrap it so that its information can be accesse
# in a standard way, both in Python space and C space.
#
# Heavily based on an idea of Andrew Straw. See
# http://www.scipy.org/Cookbook/ArrayStruct_and_Pyrex
# Very inspiring! :-)
#
# First version: 2006-03-25
# Last update: 2006-03-25
# Author: Francesc Altet
```

```
import sys

cdef extern from "Python.h":
    ctypedef int Py_intptr_t
    long PyInt_AsLong(object)
    void Py_INCREF(object)
    void Py_DECREF(object)
    object PyCObject_FromVoidPtrAndDesc(void* cobj, void* desc,
                                        void (*destr)(void *, void

cdef extern from "stdlib.h":
    ctypedef long size_t
    ctypedef long intptr_t
    void *malloc(size_t size)
    void free(void* ptr)

# for PyArrayInterface:
CONTIGUOUS=0x01
FORTRAN=0x02
ALIGNED=0x100
NOTSWAPPED=0x200
WRITEABLE=0x400

# byteorder dictionary
byteorder = {'<':'little', '>':'big'}

ctypedef struct PyArrayInterface:
    int version          # contains the integer 2 as a sanity check
    int nd               # number of dimensions
    char typekind        # kind in array --- character code of type
    int itemsize         # size of each element
    int flags            # flags indicating how the data should be
    Py_intptr_t *shape   # A length-nd array of shape information
    Py_intptr_t *strides # A length-nd array of stride information
    void *data           # A pointer to the first element of the ar

cdef void free_array_interface(void *ptr, void *arr):
    arrpy = <object>arr
    Py_DECREF(arrpy)

cdef class NumInd:
    cdef void *data
    cdef int _nd
    cdef Py_intptr_t *_shape, *_strides
    cdef PyArrayInterface *inter
    cdef object _t_shape, _t_strides, _undarray

    def __init__(self, object undarray):
        cdef int i, stride
        cdef object array_shape, array_strides

        # Keep a reference to the underlying object
```

```
        self._undarray = undarray
        # Get the shape and strides C arrays
        array_shape = undarray.__array_shape__
        self._t_shape = array_shape
        # The number of dimensions
        self._nd = len(array_shape)
        # The shape
        self._shape = <Py_intptr_t *>malloc(self._nd*sizeof(Py_intp
        for i from 0 <= i < self._nd:
            self._shape[i] = self._t_shape[i]
        # The strides (compute them if needed)
        array_strides = undarray.__array_strides__
        self._t_strides = array_strides
        self._strides = <Py_intptr_t *>malloc(self._nd*sizeof(Py_ir
        if array_strides:
            for i from 0 <= i < self._nd:
                self._strides[i] = array_strides[i]
        else:
            # strides is None. Compute them explicitely.
            self._t_strides = [0] * self._nd
            stride = int(self.typestr[2:])
            for i from self._nd > i >= 0:
                self._strides[i] = stride
                self._t_strides[i] = stride
                stride = stride * array_shape[i]
            self._t_strides = tuple(self._t_strides)
        # Populate the C array interface
        self.inter = self._get_array_interface()

    # Properties. This are visible from Python space.
    # Add as many as you want.

    property undarray:  # Returns the underlying array
        def __get__(self):
            return self._undarray

    property shape:
        def __get__(self):
            return self._t_shape

    property strides:
        def __get__(self):
            return self._t_strides

    property typestr:
        def __get__(self):
            return self._undarray.__array_typestr__

    property readonly:
        def __get__(self):
            return self._undarray.__array_data__[1]

    property __array_struct__:
```

```
        "Allows other numerical packages to obtain a new object."
        def __get__(self):
            if hasattr(self._undarray, "__array_struct__"):
                return self._undarray.__array_struct__
            else:
                # No an underlying array with __array_struct__
                # Deliver an equivalent PyCObject.
                Py_INCREF(self)
                return PyCObject_FromVoidPtrAndDesc(<void*>self.int
                                                    <void*>self,
                                                    free_array_inte

    cdef PyArrayInterface *_get_array_interface(self):
        "Populates the array interface"
        cdef PyArrayInterface *inter
        cdef object undarray, data_address, typestr

        undarray = self._undarray
        typestr = self.typestr
        inter = <PyArrayInterface *>malloc(sizeof(PyArrayInterface)
        if inter is NULL:
            raise MemoryError()

        inter.version = 2
        inter.nd = self._nd
        inter.typekind = ord(typestr[1])
        inter.itemsize = int(typestr[2:])
        inter.flags = 0  # initialize flags
        if typestr[0] == '|':
            inter.flags = inter.flags | NOTSWAPPED
        elif byteorder[typestr[0]] == sys.byteorder:
            inter.flags = inter.flags | NOTSWAPPED
        if not self.readonly:
            inter.flags = inter.flags | WRITEABLE
        # XXX how to determine the ALIGNED flag?
        inter.strides = self._strides
        inter.shape = self._shape
        # Get the data address
        data_address = int(undarray.__array_data__[0], 16)
        inter.data = <void*>PyInt_AsLong(data_address)
        return inter

    # This is just an example on how to modify the data in C space
    # (and at C speed! :-)
    def modify(self):
        "Modify the values of the underlying array"
        cdef int *data, i

        data = <int *>self.inter.data
        # Modify just the first row
        for i from 0 <= i < self.inter.shape[self.inter.nd-1]:
            data[i] = data[i] + 1
```

```
    def __dealloc__(self):
        free(self._shape)
        free(self._strides)
        free(self.inter)
```

An example of use

In order to get an idea of what the above extension offers, try to run this script against the !NumInd extension:

```
import Numeric
import numarray
import numpy
import numind

# Create an arbitrary object for each package
nu=Numeric.arange(12)
nu.shape = (4,3)
na=numarray.arange(12)
na.shape = (4,3)
np=numpy.arange(12)
np.shape = (4,3)

# Wrap the different objects with the NumInd class
# and execute some actions on it
for obj in [nu, na, np]:
    ni = numind.NumInd(obj)
    print "original object type-->", type(ni.undarray)
    # Print some values
    print "typestr -->", ni.typestr
    print "shape -->", ni.shape
    print "strides -->", ni.strides
    npa = numpy.asarray(ni)
    print "object after a numpy re-wrapping -->", npa
    ni.modify()
    print "object after modification in C space -->", npa
```

You can check the output here `test_output.txt`

See also

- ["Cookbook/Pyrex_and_NumPy"]
- ["Cookbook/ArrayStruct_and_Pyrex"] (The inspiring recipe)

Attachments

- `test_output.txt`

Array, struct, and Pyrex

NOTE: this entry was last updated 2007-11-18, and may contain information that is not as relevant today (as of 2013).

Overview

Pyrex is a language for writing C extensions to Python. Its syntax is very similar to writing Python. A file is compiled to a file, which is then compiled like a standard C extension module for Python. Many people find writing extension modules with Pyrex preferable to writing them in C or using other tools, such as SWIG.

See http://numeric.scipy.org/ for an explanation of the interface. The following packages support the interface:

* !NumPy (all versions) * Numeric (>=24.2) * numarray (>=1.5.0)

Sharing data malloced by a Pyrex-based extension

Here is a Pyrex file which shows how to export its data using the array interface. This allows various Python types to have a view of the data without actually copying the data itself.

```
cdef extern from "Python.h":
    ctypedef int Py_intptr_t
    void Py_INCREF(object)
    void Py_DECREF(object)
    object PyCObject_FromVoidPtrAndDesc( void* cobj, void* desc, vo

cdef extern from "stdlib.h":
    ctypedef int size_t
    ctypedef long intptr_t
    void *malloc(size_t size)
    void free(void* ptr)

# for PyArrayInterface:
CONTIGUOUS=0x01
FORTRAN=0x02
ALIGNED=0x100
NOTSWAPPED=0x200
WRITEABLE=0x400

ctypedef struct PyArrayInterface:
    int version          # contains the integer 2 as a sanity check
    int nd               # number of dimensions
    char typekind        # kind in array --- character code of type
```

```
    int itemsize         # size of each element
    int flags            # flags indicating how the data should be
    Py_intptr_t *shape   # A length-nd array of shape information
    Py_intptr_t *strides # A length-nd array of stride information
    void *data           # A pointer to the first element of the ar

cdef void free_array_interface( void* ptr, void *arr ):
    cdef PyArrayInterface* inter

    inter = <PyArrayInterface*>ptr
    arrpy = <object>arr
    Py_DECREF( arrpy )
    free(inter)

ctypedef unsigned char fi
ctypedef fi* fiptr
cdef class Unsigned8Buf:
    cdef fiptr data
    cdef Py_intptr_t shape[2]
    cdef Py_intptr_t strides[2]

    def __init__(self, int width, int height):
        cdef int bufsize
        bufsize = width*height*sizeof(fi)
        self.data=<fiptr>malloc( bufsize )
        if self.data==NULL: raise MemoryError("Error allocating mem
        self.strides[0]=width
        self.strides[1]=1 # 1 byte per element

        self.shape[0]=height
        self.shape[1]=width

    def __dealloc__(self):
            free(self.data)

    property __array_struct__:
        def __get__(self):
            cdef PyArrayInterface* inter

            cdef Py_intptr_t *newshape   # A length-nd array of sha
            cdef Py_intptr_t *newstrides # A length-nd array of str
            cdef int nd

            nd = 2

            inter = <PyArrayInterface*>malloc( sizeof( PyArrayInter
            if inter is NULL:
                raise MemoryError()

            inter.version = 2
            inter.nd = nd
            inter.typekind = 'u'[0] # unsigned int
            inter.itemsize = 1
```

```
            inter.flags = NOTSWAPPED | ALIGNED | WRITEABLE
            inter.strides = self.strides
            inter.shape = self.shape
            inter.data = self.data
            Py_INCREF(self)
            obj = PyCObject_FromVoidPtrAndDesc( <void*>inter, <voic
            return obj
```

Using data malloced elsewhere with a Pyrex-based extension

One example is the get_data_copy() function of *cam*iface_shm.pyx in the motmot camera utilities software. In this use example, an image is copied from shared memory into an externally malloced buffer supporting the {{{**array_struct**}}} interface. (The shared memory stuff has only been tested on linux, but the rest should work anywhere.)

See also

- [“Cookbook/Pyrex_and_NumPy”]

Data Frames

> **NOTE**: this entry is outdated. You will likely want to use pandas instead.

The `DataFrame.py` class, posted by Andrew Straw on the) scipy-user mailing list original link, is an extremely useful tool for using alphanumerical tabular data, as often found in databases. Some data which might be ingested into a data frame could be: || **ID** || **LOCATION** || **VAL_1** || **VAL_2** || || 01 || Somewhere || 0.1 || 0.6 || || 02 || Somewhere Else || 0.2 || 0.5 || || 03 || Elsewhere || 0.3 || 0.4 ||

The `DataFrame.py` class can be populated from data from a CSV file (comman-separated values). In its current implementation, these files are read with Python's own CSV module, which allows for a great deal of customisation.

Example Usage

A sample file CSV file from Access2000 is in `CSVSample.csv` We first import the module:

```
import DataFrame
```

and read the file in using our desired CVS dialect:

```
df = DataFrame.read_csv("CSVSample.csv",dialect=DataFrame.access20(
```

(note that the dialect is actually defined in the DataFrame class). It is often useful to filter the data according to some criterion.

Compatibility with Python 2.6 and above

Starting with Python 2.6, the sets module is deprecated, in order to get rid of the warning, replace

```
imports sets
```

with

```
try:
    set
except NameError:
    from sets import Set as set
```

Attachments

- `CSVSample.csv`
- `DataFrame.py`
- `SampleCSV.csv`

Python Imaging Library

```
(!) see also [:Cookbook/Matplotlib/LoadImage] to load a PNG image
```

Apply this patch to make PIL Image objects both export and consume the array interface (from Travis Oliphant):

```
Index: PIL/Image.py
===================================================================
--- PIL/Image.py    (revision 358)
+++ PIL/Image.py    (working copy)
@@ -187,6 +187,42 @@

 }

+if sys.byteorder == 'little':
+    _ENDIAN = '<'
+else:
+    _ENDIAN = '>'
+
+_MODE_CONV = {
+
+    # official modes
+    "1": ('|b1', None),
+    "L": ('|u1', None),
+    "I": ('%si4' % _ENDIAN, None),
+    "F": ('%sf4' % _ENDIAN, None),
+    "P": ('|u1', None),
+    "RGB": ('|u1', 3),
+    "RGBX": ('|u1', 4),
+    "RGBA": ('|u1', 4),
+    "CMYK": ('|u1', 4),
+    "YCbCr": ('|u1', 4),
+
+    # Experimental modes include I;16, I;16B, RGBa, BGR;15,
+    # and BGR;24\.  Use these modes only if you know exactly
+    # what you're doing...
+
+}
+
+def _conv_type_shape(im):
+    shape = im.size[::-1]
+    typ, extra = _MODE_CONV[im.mode]
+    if extra is None:
+        return shape, typ
+    else:
+        return shape+(extra,), typ
+
+
```

```
+
+
 MODES = _MODEINFO.keys()
 MODES.sort()

@@ -491,6 +527,22 @@
         return string.join(data, "")

     ##
+    # Returns the array_interface dictionary
+    #
+    # @return A dictionary with keys 'shape', 'typestr', 'data'
+
+    def __get_array_interface__(self):
+        new = {}
+        shape, typestr = _conv_type_shape(self)
+        new['shape'] = shape
+        new['typestr'] = typestr
+        new['data'] = self.tostring()
+        return new
+
+    __array_interface__ = property(__get_array_interface__, None,
+
+
+    ##
     # Returns the image converted to an X11 bitmap.  This method
     # only works for mode "1" images.
     #
@@ -1749,7 +1801,61 @@

     return apply(fromstring, (mode, size, data, decoder_name, args

+
 ##
+# (New in 1.1.6) Create an image memory from an object exporting
+#  the array interface (using the buffer protocol).
+#
+#  If obj is not contiguous, then the tostring method is called
+#  and frombuffer is used
+#
+# @param obj Object with array interface
+# @param mode Mode to use (will be determined from type if None)
+
+def fromarray(obj, mode=None):
+    arr = obj.__array_interface__
+    shape = arr['shape']
+    ndim = len(shape)
+    try:
+        strides = arr['strides']
+    except KeyError:
+        strides = None
+    if mode is None:
+        typestr = arr['typestr']
```

```
+        if not (typestr[0] == '|' or typestr[0] == _ENDIAN or
+                typestr[1:] not in ['u1', 'b1', 'i4', 'f4']):
+            raise TypeError, "cannot handle data-type"
+        typestr = typestr[:2]
+        if typestr == 'i4':
+            mode = 'I'
+        elif typestr == 'f4':
+            mode = 'F'
+        elif typestr == 'b1':
+            mode = '1'
+        elif ndim == 2:
+            mode = 'L'
+        elif ndim == 3:
+            mode = 'RGB'
+        elif ndim == 4:
+            mode = 'RGBA'
+        else:
+            raise TypeError, "Do not understand data."
+    ndmax = 4
+    bad_dims=0
+    if mode in ['1','L','I','P','F']:
+        ndmax = 2
+    elif mode == 'RGB':
+        ndmax = 3
+    if ndim > ndmax:
+        raise ValueError, "Too many dimensions."
+
+    size = shape[:2][::-1]
+    if strides is not None:
+        obj = obj.tostring()
+
+    return frombuffer(mode, size, obj)
+
+##
 # Opens and identifies the given image file.
 # <p>
 # This is a lazy operation; this function identifies the file, but
```

Exemple

```
>>> import Image
>>> im=Image.open('foo1.png')
>>> a=numpy.array(p)
# do something with a ...
>>> im = Image.fromarray(a)
>>> im.save( 'foo2.png' )
```

Recipes for timeseries

NOTE: The official documentation and important remarks from the developers can be found at the timseries scikit sourceforge page.

The FortranFile class

> **NOTE**: you may want to use scipy.io.FortranFile instead.

This subclass of file is designed to simplify reading of Fortran unformatted binary files which are typically saved in a record-based format.

```
# Copyright 2008, 2009 Neil Martinsen-Burrell
#
# Permission is hereby granted, free of charge, to any person obtai
# of this software and associated documentation files (the "Softwar
# in the Software without restriction, including without limitatior
# to use, copy, modify, merge, publish, distribute, sublicense, and
# copies of the Software, and to permit persons to whom the Softwar
# furnished to do so, subject to the following conditions:

# The above copyright notice and this permission notice shall be ir
# all copies or substantial portions of the Software.

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, E
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTA
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, A
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEA
# THE SOFTWARE.

"""Defines a file-derived class to read/write Fortran unformatted f

The assumption is that a Fortran unformatted file is being written
the Fortran runtime as a sequence of records.  Each record consists
an integer (of the default size [usually 32 or 64 bits]) giving the
length of the following data in bytes, then the data itself, then t
same integer as before.

Examples
--------

To use the default endian and size settings, one can just do::
 >>> f = FortranFile('filename')
 >>> x = f.readReals()

One can read arrays with varying precisions::
 >>> f = FortranFile('filename')
 >>> x = f.readInts('h')
 >>> y = f.readInts('q')
 >>> z = f.readReals('f')
Where the format codes are those used by Python's struct module.
```

```
One can change the default endian-ness and header precision::
 >>> f = FortranFile('filename', endian='>', header_prec='l')
for a file with little-endian data whose record headers are long
integers.
"""

__docformat__ = "restructuredtext en"

import struct
import numpy

class FortranFile(file):

    """File with methods for dealing with fortran unformatted data

    def _get_header_length(self):
        return struct.calcsize(self._header_prec)
    _header_length = property(fget=_get_header_length)

    def _set_endian(self,c):
        """Set endian to big (c='>') or little (c='<') or native (c

 :Parameters:
 `c` : string
 The endian-ness to use when reading from this file.
 """
        if c in '<>@=':
            self._endian = c
        else:
            raise ValueError('Cannot set endian-ness')
    def _get_endian(self):
        return self._endian
    ENDIAN = property(fset=_set_endian,
                      fget=_get_endian,
                      doc="Possible endian values are '<', '>', '@
                     )

    def _set_header_prec(self, prec):
        if prec in 'hilq':
            self._header_prec = prec
        else:
            raise ValueError('Cannot set header precision')
    def _get_header_prec(self):
        return self._header_prec
    HEADER_PREC = property(fset=_set_header_prec,
                           fget=_get_header_prec,
                           doc="Possible header precisions are 'h',
                          )

    def __init__(self, fname, endian='@', header_prec='i', *args, *
        """Open a Fortran unformatted file for writing.

 Parameters
```

```
----------
endian : character, optional
Specify the endian-ness of the file.  Possible values are
'>', '<', '@' and '='.  See the documentation of Python's
struct module for their meanings.  The deafult is '>' (native
byte order)
header_prec : character, optional
Specify the precision used for the record headers.  Possible
values are 'h', 'i', 'l' and 'q' with their meanings from
Python's struct module.  The default is 'i' (the system's
default integer).

"""
       file.__init__(self, fname, *args, **kwargs)
       self.ENDIAN = endian
       self.HEADER_PREC = header_prec

   def _read_exactly(self, num_bytes):
       """Read in exactly num_bytes, raising an error if it can't
       data = ''
       while True:
           l = len(data)
           if l == num_bytes:
               return data
           else:
               read_data = self.read(num_bytes - l)
           if read_data == '':
               raise IOError('Could not read enough data.'
                             '  Wanted %d bytes, got %d.' % (num_b
           data += read_data

   def _read_check(self):
       return struct.unpack(self.ENDIAN+self.HEADER_PREC,
                            self._read_exactly(self._header_length
                           )[0]

   def _write_check(self, number_of_bytes):
       """Write the header for the given number of bytes"""
       self.write(struct.pack(self.ENDIAN+self.HEADER_PREC,
                              number_of_bytes))

   def readRecord(self):
       """Read a single fortran record"""
       l = self._read_check()
       data_str = self._read_exactly(l)
       check_size = self._read_check()
       if check_size != l:
           raise IOError('Error reading record from data file')
       return data_str

   def writeRecord(self,s):
       """Write a record with the given bytes.
```

```
Parameters
----------
s : the string to write

"""
        length_bytes = len(s)
        self._write_check(length_bytes)
        self.write(s)
        self._write_check(length_bytes)

    def readString(self):
        """Read a string."""
        return self.readRecord()

    def writeString(self,s):
        """Write a string

Parameters
----------
s : the string to write

"""
        self.writeRecord(s)

    _real_precisions = 'df'

    def readReals(self, prec='f'):
        """Read in an array of real numbers.

Parameters
----------
prec : character, optional
Specify the precision of the array using character codes from
Python's struct module.  Possible values are 'd' and 'f'.

"""

        _numpy_precisions = {'d': numpy.float64,
                             'f': numpy.float32
                            }

        if prec not in self._real_precisions:
            raise ValueError('Not an appropriate precision')

        data_str = self.readRecord()
        num = len(data_str)/struct.calcsize(prec)
        numbers =struct.unpack(self.ENDIAN+str(num)+prec,data_str)
        return numpy.array(numbers, dtype=_numpy_precisions[prec])

    def writeReals(self, reals, prec='f'):
        """Write an array of floats in given precision

Parameters
```

```
----------
reals : array
Data to write
prec` : string
Character code for the precision to use in writing
"""
        if prec not in self._real_precisions:
            raise ValueError('Not an appropriate precision')

        # Don't use writeRecord to avoid having to form a
        # string as large as the array of numbers
        length_bytes = len(reals)*struct.calcsize(prec)
        self._write_check(length_bytes)
        _fmt = self.ENDIAN + prec
        for r in reals:
            self.write(struct.pack(_fmt,r))
        self._write_check(length_bytes)

    _int_precisions = 'hilq'

    def readInts(self, prec='i'):
        """Read an array of integers.

Parameters
----------
prec : character, optional
Specify the precision of the data to be read using
character codes from Python's struct module.  Possible
values are 'h', 'i', 'l' and 'q'

"""
        if prec not in self._int_precisions:
            raise ValueError('Not an appropriate precision')

        data_str = self.readRecord()
        num = len(data_str)/struct.calcsize(prec)
        return numpy.array(struct.unpack(self.ENDIAN+str(num)+prec,

    def writeInts(self, ints, prec='i'):
        """Write an array of integers in given precision

Parameters
----------
reals : array
Data to write
prec : string
Character code for the precision to use in writing
"""
        if prec not in self._int_precisions:
            raise ValueError('Not an appropriate precision')

        # Don't use writeRecord to avoid having to form a
        # string as large as the array of numbers
```

```
        length_bytes = len(ints)*struct.calcsize(prec)
        self._write_check(length_bytes)
        _fmt = self.ENDIAN + prec
        for item in ints:
            self.write(struct.pack(_fmt,item))
        self._write_check(length_bytes)
```

dbase

> **NOTE**: You may want to use pandas instead of this.

The `dbase.py` class, can be used to read/write/summarize/plot time-series data.

To summarize the functionality:

1. data and variable names stored in a dictionary - accessible using variable names
2. load/save from/to csv/pickle format, including date information (shelve format to be added)
3. plotting and descriptive statistics, with dates if provided
4. adding/deleting variables, including trends/(seasonal)dummies
5. selecting observations based on dates or other variable values (e.g., > 1/1/2003)
6. copying instance data

Attached also the `dbase_pydoc.txt` information for the class.

Example Usage

To see the class in action download the file and run it (python dbase.py). This will create an example data file (./dbase_test_files/data.csv) that will be processed by the class.

To import the module:

```
import sys
sys.path.append('attachments/dbase')
import dbase
```

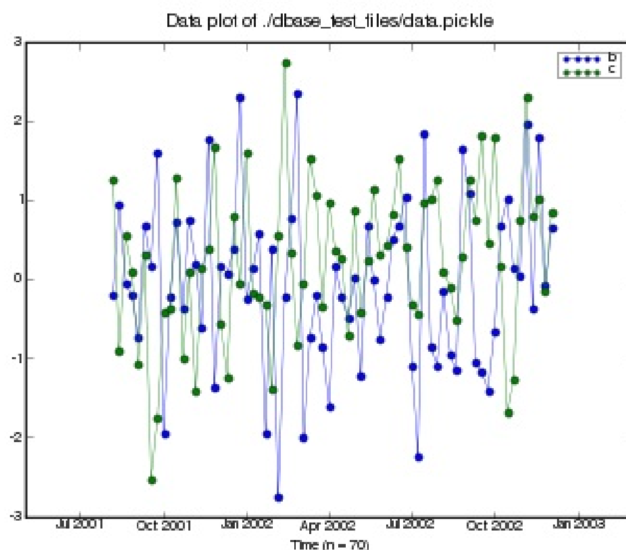
After running the class you can load the example data using

```
data = dbase.dbase("attachments/dbase/data.csv", date = 0)
```

In the above command ‘0’ is the index of the column containing dates.

You can plot series ‘b’ and ‘c’ in the file using

```
data.dataplot('b','c')
```



You get descriptive statistics for series 'a','b', and 'c' by using

```
data.info('a','b','c')
```

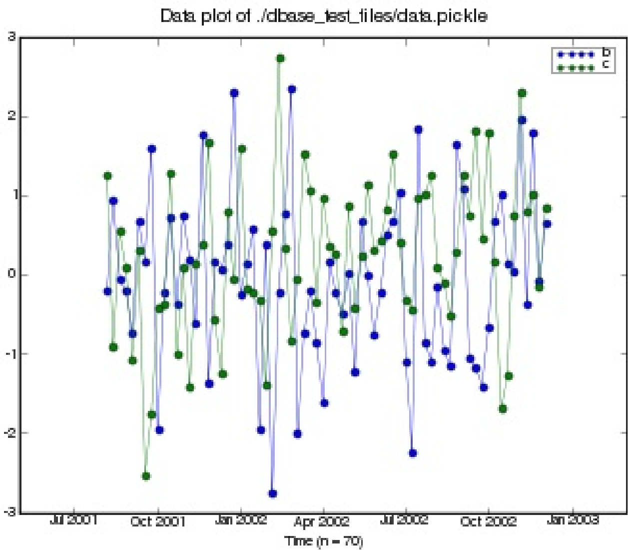
```
===================================================================
============================= Database information ================
===================================================================

file:                                    /mnt/data/pauli/prj/scipy/SciP
# obs:                                   100
# variables:                     3
Start date:                      08 Jan 2001
End date:                        02 Dec 2002

var                              min                       max
===================================================================
a                                -2.56            3.35            -0.08
b                                -2.00            2.16            -0.02
c                                -1.91            2.54             0.18
```

Attachments

- data.csv
- dbase.py
- dbase_pydoc.0.2.txt
- ex_plot.0.1.png


Data plot of ./dbase_test_files/data.pickle
b
c
3
2
1
0
-1
-2
-3
Jul 2001
Oct 2001
Jan 2002
Apr 2002
Jul 2002
Oct 2002
Jan 2003
Time (n = 70)

xplt

This shows a simple example of how to create a quick 3-d surface visualization using xplt.

```
from scipy.sandbox import xplt
from numpy import *
from scipy import special

x,y = ogrid[-12:12:50j,-12:12:50j]
r = sqrt(x**2+y**2)
z = special.j0(r)
xplt.surf(z,x,y,shade=1,palette='heat')
```

Attachments

- `surface.png`

